BACHELORARBEIT

Herr
Sergius Beller

Analyse und Entwurf eines RESTful Web Services für eine Cluster-Managementsoftware

2011

Fakultät **Mathematik/Naturwissenschaften/Informatik**

BACHELORARBEIT

Analyse und Entwurf eines RESTful Web Services für eine Cluster-Managementsoftware

Autor:
Sergius Beller

Studiengang:
Informatik, Bachelor

Seminargruppe:
IF08w1B

Betreuer:
Prof. Dr.-Ing. habil. Joachim Geiler, Hochschule Mittweida

Betreuer:
Dipl.-Math. techn. Robert Hommel, MEGWARE Computer GmbH

Mittweida, Dezember 2011

Bibliografische Angaben

Beller, Sergius: Analyse und Entwurf eines RESTful Web Services für eine Cluster-Managementsoftware, 75 Seiten, 15 Abbildungen, Hochschule Mittweida (FH), Fakultät Mathematik/Naturwissenschaften/Informatik

Bachelorarbeit, 2011

Satz: LaTeX

Referat

In der vorliegenden Bachelorarbeit wird ein Web Service API für die Megware Cluster-Managementsoftware entworfen und ansatzweise implementiert. Der REST-Architekturstil bildet dabei das Fundament des Entwurfes, auf dem das Web Service aufgebaut wird. Einen großen Stellenwert nehmen die Performance-Betrachtungen ein, um die internen Einflüsse wie Lastverteilung und Caching zu berücksichtigen.

I. Inhaltsverzeichnis

II. Abbildungs- und Tabellenverzeichnis

Abbildungen

Tabellen

III. Abkürzungsverzeichnis

AJAX	Asynchronous JavaScript and XML
API	Application Programming Interface
CLI	Command-Line Interface
CRUD	Create, Read, Update, Delete
CSV	Comma-Separated Values
DB	Datenbank
DRY	Don't repeat yourself
EER	Enhanced Entity-Relationship Model
ERB	Embedded Ruby
GUI	Graphical User Interface
HA	High Availability
Haml	HTML Abstraction Markup Language
HPC	High-Performance Computing
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
IPMI	Intelligent Platform Management Interface
JS	JavaScript
JSON	JavaScript Object Notation
JSP	JavaServer Pages
MVC	Model-View-Controller
ORM	Object-Relational-Mapping
PDF	Portable Document Format
PDU	Power Distribution Unit
RAID	Redundant Array of Independent Disks
RDF	Resource Description Framework
REST	Representational State Transfer
RFC	Request for Comments
ROA	Resource-Oriented Architecture
RoR	Ruby on Rails
RPC	Remote Procedure Call
RRD	Round-Robin-Database

Sass	Syntactically Awesome Stylesheets
SLB	Server Load Balancing
SNMP	Simple Network Management Protocol
SSH	Secure Shell
SSL	Secure Sockets Layer
TCP	Transmission Control Protocol
TDD	Test-Driven Development
TLS	Transport Layer Security
UDP	User Datagram Protocol
URI	Uniform Resource Identifier
WS	Web Service
WWW	World Wide Web
XHTML	Extensible Hypertext Markup Language
XML	Extensible Markup Language
YAML	Yet Another Multicolumn Layout

1 Einleitung

Die Hochleistungsrechner sind heutzutage aus unserer Welt nicht mehr wegzudenken. Sie unterstützen die Wissenschaftler auf den Gebieten der Biologie, Chemie, Geologie, Klimaforschung, Luft- und Raumfahrt, Medizin und Physik. Für eine einfache Steuerung und Konfiguration der Hochleistungsrechner werden Cluster-Managementsysteme gebraucht, die den Verwaltungsaufwand erheblich minimieren können. Eine zukunftsorientierte Architektur der Managementsoftware spielt dabei eine entscheidende Rolle.

1.1 Motivation

Die Idee zur Entwicklung einer REST-basierter Architektur für Cluster-Managementsoftware entstand aus dem Praktikum bei der Megware Computer GmbH. Die Hauptaufgabe dieses Praktikums bestand in der Analyse des Qt-Frameworks für die Entwicklung einer grafischen Oberfläche für die Megware Cluster-Managementsoftware.

Das Qt-Framework diente in diesem Zusammenhang nicht nur dem Zweck der Entwicklung einer Benutzeroberfläche, sondern auch als Basis für die Netzwerk- und Kommunikationsschicht. Das ergab sich daraus, dass für die Cluster-Managementsoftware zu diesem Zeitpunkt kein API für netzwerkweite Kommunikation zur Verfügung stand.

Im Zuge der Entwicklung offenbarte das Qt als Netzwerk-Framework jedoch einige Schwächen. Zum einen war nur die Low-Level-Programmierung auf der Socket-Ebene möglich, zum anderen sollte das aktuellste Qt-Framework auf allen Zielplattformen zur Verfügung stehen, was auf den mobilen Betriebssystemen nur bedingt möglich ist.[1]

Ein weiterer Grund, das Qt-Framework nicht für die Netzwerkprogrammierung einzusetzen, bestand in einer mangelhaften Anbindung der Programmiersprache Ruby 1.9 an das Qt-Framework 4.7. Die Annahme, man könne den Qt-Backend auch für die in Ruby entwickelte Weboberfläche verwenden, ist somit nicht ganz korrekt.

Als Lösung dieses Problems bot sich der Einsatz von RESTful Web Services basierend auf dem Framework Ruby on Rails an. Da dieses Framework bereits eine Anwendung bei der Megware Computer GmbH gefunden hat und der REST-Ansatz sehr aktuell ist, stellt dieses Thema eine besondere Herausforderung als Gegenstand der Bachelorarbeit dar.

[1] Qt-Framework steht für viele mobile Betriebssysteme zur Verfügung, so wird die Entwicklung unter Symbian, Maemo/MeeGo, Windows CE und Windows Mobile direkt unterstützt, vgl. [36]. Für andere weit verbreitete mobile Plattformen wie Android und Apple iOS gibt es keine Unterstützung seitens Nokia, sondern nur Community-Portierungen im Alpha-Stadium, vgl. [3] und [38]

1.2 Problemstellung

Da die Cluster-Managementsoftware kein produktionsreifes API besitzt, ergeben sich folgende Probleme:

- direkte Kommunikation mit dem Appliance-Daemon über das Appliance CLI verursacht viele überflüssige Anfragen;
- jeder Client (Weboberfläche, Qt-GUI, TFT-Bedienpanel RackView) implementiert einen eigenen Parser, dies erhöht die Komplexität und Fehleranfälligkeit der Endanwendungen;
- flache Lernkurve erschwert die Entwicklung weiterer Anwendungen, die auf die Daten der Cluster-Managementsoftware zugreifen.

Der Entwurf eines neuen API für die Megware Cluster-Managementsoftware soll somit die Defizite bei der Client-Server-Kommunikation beseitigen.

Da das Qt-Framework eine plattformunabhängige Kommunikation nur sehr bedingt ermöglicht, soll eine neue Architektur erarbeitet werden, welche leicht an die bestehenden Lösungen anpassbar und durch neue Module erweiterbar ist. Dadurch soll die Entwicklung leichter, moderner und attraktiver werden.

1.3 Zielsetzung

In dieser Arbeit soll ein Web Service API ausgearbeitet werden, welches auf der REST-Architektur basiert. Ferner soll überprüft werden, inwieweit diese Architektur für die API-Entwicklung geeignet ist und wie hoch ihre Leistungsfähigkeit ist.

Hierbei sollten insbesondere einzelne Programmschichten wie Parser, Modell und Controller ansatzweise implementiert werden. Idealerweise soll ein bestimmter Teil des Systems durch alle Ebenen hindurch implementiert werden, um die Qualitäten wie Performance oder Datendurchsatz durch eine Anbindung an die Qt-Oberfläche praxisnah zu überprüfen. Eine umfangreichere Funktionalität soll das Programm später erweitern.[2]

Es muss bemerkt werden, dass die Performance eines auf dem Web Service basierten API kaum an die Performance einer Qt-basierten Lösung heranreichen kann. Trotzdem ist dieser Nachteil durchaus vertretbar, da dadurch eine bessere Plattformunabhängigkeit und Wartbarkeit erreicht werden kann.

Ein weiterer wichtiger Faktor ist der Durchsatz. Damit die zu erarbeitende Lösung auch mit steigender Last nicht kollabiert, soll die neue Architektur gleich von Anfang an auf eine höhere Skalierbarkeit ausgelegt werden.

[2] Siehe Qualitätsmerkmale nach ISO 9126 im Anhang A.

1.4 Methodisches Vorgehen

Die vorliegende Arbeit besteht aus einem theoretischen und einem praktischen Teil. In dem theoretischen Teil wird eine Literaturrecherche durchgeführt und eine einheitliche Begriffsbasis für die Technologien REST und Ruby on Rails geschaffen. Weiterhin sorgt ein kurzer Überblick zu den Themen Cluster und Cluster-Managementsysteme für ein besseres Verständnis der Problematik dieser Arbeit.

In dem praktischen Teil wird die Megware Cluster-Managementsoftware näher betrachtet und ausschnittsweise analysiert. Anschließend sind ein erster Entwurf des RESTful Web Services zu entwickeln und verschiedene Skalierungsmöglichkeiten zu diskutieren. Wichtige Sicherheitsaspekte sollen dabei betrachtet und zum Teil implementiert werden.

Abschließend wird auf die erstellten Prototypen eingegangen und die Benchmarks werden ausgewertet. Zuletzt werden die gewonnene Erkenntnisse zusammengefasst und die Aussichten vorgestellt.

2 REST-Architekturstil

Motiviert durch den technischen Erfolg des WWW, wurde der Begriff REST (Representational State Transfer) erstmals von Roy Fielding in seiner Doktorarbeit aus dem Jahr 2000 formuliert. In dieser Arbeit versucht der Autor die wichtigsten architektonischen Prinzipien des WWW zu verstehen und zu erfassen, um diese dann in web-ähnlichen Architekturen richtig anwenden zu können.[3]

Allgemein versteht man unter REST einen abstrakten Architekturstil, welcher beschreibt, wie die Standards in einer gerechten Weise für die Entwicklung von verteilten Hypermedia-Systemen eingesetzt werden sollen.[4]

REST ist kein Standard bzw. keine Technologie. Man sollte REST als eine Anleitung bzw. Referenz für die Entwicklung von Webanwendungen betrachten. Die Architekturen, die sich an dem REST-Architekturstil orientieren und deren Regeln befolgen, werden somit als RESTful bezeichnet. Als RESTful Web Service wird ein eindeutig identifizierbares Software-System bezeichnet, welches die Interoperabilität zwischen den verschiedenen Anwendungen im Netzwerk unterstützt, indem die Ressourcen durch deren Repräsentationen mit einer einheitlichen Menge der Standard-Operationen manipuliert werden.[5]

Obwohl die Prinzipien von REST auch mit anderen Protokollen umgesetzt werden können, sind nur das WWW mit dem HTTP-Protokoll und der URI als Implementierung praxisrelevant.[6] HTTP ist das wichtigste Anwendungsprotokoll im WWW, welches die Kommunikation zwischen Webserver und Webbrowser im Intranet bzw. Internet ermöglicht.[7]

2.1 Warum REST?

REST bietet einen einfachen Weg komplexe und unabhängige Systeme miteinander zu verbinden. Die Popularität von REST wächst seit 2005 ständig, was auch die zahlreichen Web Services von Google, Yahoo, Bing, Twitter und Flickr belegen. Das liegt daran, dass REST direkt auf dem HTTP-Protokoll basiert und selbst kein zusätzliches Protokoll bzw. Standard einführt und damit den zusätzlichen Overhead vermeidet. So kann REST überall dort angewendet werden, wo auch HTTP eingesetzt werden kann.[8]

[3] Vgl. [15], [46] und [14] Conclusions.
[4] Vgl. [6] und [15].
[5] Vgl. [52], [55] Kapitel 1.4 What is a Web service und Kapitel 3.1.3 Relationship to the WWW and REST Architectures.
[6] Vgl. [6].
[7] Vgl. [22].
[8] Vgl. [15].

Bei manchen kritischen Anwendungen kann das HTTP-Protokoll nicht die nötige Effizienz anbieten, hier sollte man evtl. auf CORBA, RMI oder AMQP ausweichen.[9]

2.2 Grundlagen

REST kann mit 5 folgenden Kernprinzipien charakterisiert werden: [10]

- eindeutig identifizierbare Ressourcen,
- Repräsentationen,
- Hypermedia,
- Standardmethoden,
- statuslose Kommunikation.

Unter einer **Ressource** kann ein Objekt vorgestellt werden. Die Ressource soll eindeutig identifizierbar sein und mindestens eine Repräsentation (Darstellung einer Ressource) besitzen. Ressourcen bilden die zentrale Idee in REST.[11]

Repräsentationen sind also Darstellungen einer Ressource. So kann ein Objekt verschiedene Repräsentationen besitzen, z. B. in Formaten XHTML, XML, JSON oder PDF. Alle Repräsentationen sind gleichermaßen gültig.[12]

Hypermedia ist ein zentrales Konzept der Anwendungssteuerung. Unter diesem Konzept verbergen sich die Idee der Verknüpfung zwischen den Ressourcen und die Idee der Steuerung des Anwendungszustands. Somit kann eine Anwendung ohne Kenntnis der Struktur eines Services nicht nur an die nötige Ressource gelangen, sondern auch den Ressourcenstatus ändern.[13]

Standardmethoden sind ein Satz von Methoden, welche alle Ressourcen unterstützen, aber nicht implementieren sollen. Hiermit wird für jede Ressource ein und dasselbe Interface verwendet, welches für alle Kommunikationspartner bekannt ist.[14]

Statuslose Kommunikation ermöglicht Skalierbarkeit und Unabhängigkeit zwischen aufeinanderfolgenden Anfragen. Der Session-Zustand bzw. Benutzerstatus soll dabei entweder vollständig auf dem Client gehalten werden oder als eine eigene Ressource auf dem Server hinterlegt werden.[15]

[9] Vgl. [6].
[10] Vgl. [53] S. 9.
[11] Vgl. [53] S. 31.
[12] Vgl. [53] S. 31 und [45] S. 81.
[13] Vgl. [53] S. 10–11 und [45] S. 94.
[14] Vgl. [53] S. 11–13.
[15] Vgl. [53] S. 15 und [45] S. 217–218.

2.3 Ressourcen

Allgemein gilt im REST alles als Ressource, was referenziert werden soll. Falls man etwas verlinken, repräsentieren, bearbeiten oder auflisten kann, dann soll dies ebenfalls als eine Ressource definiert werden.[16]

Die Ressourcen kann man mit den Objekten aus der objektorientierten Programmierung vergleichen, mit dem Unterschied, dass die Ressourcen viel langlebiger sind und deren Geltungsbereich viel größer ist.[17]

Die Definition einer Ressource ist sehr abstrakt. So können einige Ressourcen auf der Festplatte gespeichert werden, andere Ressourcen sind wiederum physischer oder abstrakter Natur und lassen sich nur bedingt speichern.[18] Hier sind einige Beispiele:

- ein Weblog-Eintrag vom 06.06.2006;
- ein SciFi-Buch „Der Wüstenplanet“ von Frank Herbert, Heyne Verlag, 2001;
- ein Flugzeug Airbus A380, Baujahr 2007.

Weiterhin können die Ressourcen in unterschiedliche Kategorien unterteilt werden: [19]

- **Primär- und Subressourcen** – die Primärressource „Kunde“ kann z. B. mit einer Subressource „Adresse“ assoziiert werden;
- **Listen und Filter** – es ist oftmals sinnvoll die Menge aller Kunden zu identifizieren, indem man eine Listenressource definiert; die Listen können auch gefiltert werden, indem die Filterkriterien festlegt werden (z. B. alle Kunden aus Chemnitz); oftmals können die Listenressourcen sehr groß werden, was mit Paginierung (seitenweise Aufteilung) gehandhabt wird;
- **Projektionen und Aggregationen** – sollen die zu übertragende Datenmengen reduziert werden, so ist es nützlich nur eine Attributuntermenge einer Ressource zu übertragen; sollen Client/Server-Interaktionen reduziert werden, so kann es angebracht sein, die Attribute verschiedener Ressourcen zusammenzufassen;
- **Aktivitäten** – sind prozessbedingte Ressourcen, z. B. wie Stornierung einer Rechnung oder Aus-/Einschalten eines Computers.

Wie bereits erwähnt, gehören die Ressourcen zu dem zentralen Konzept im REST. Beim Entwurf stehen die Ressourcen im Mittelpunkt der Betrachtungen und nicht die Methoden, wie beim objektorientierten Entwurf. Aus diesem Grund werden die RESTful Architekturen auch als ROA (Resource Oriented Architecure) bezeichnet.[20]

[16] Vgl. [45] S. 81.
[17] Vgl. [53] S. 36.
[18] Vgl. [45] S. 81.
[19] Vgl. [53] S. 32–35
[20] Vgl. [53] S. 18, 31–32 und [45] S. 79–80.

2.4 Ressourcenidentifikation

Jede Ressource muss mindestens eine eindeutige Identifikation, eine URI, besitzen. Bei einer URI (Universal Resource Identifier) handelt es sich um Name und Adresse einer Ressource. Die URI wird für die Identifikation von Ressourcen im WWW eingesetzt. Besitzt ein Teil der Information keine URI, so stellt das keine Ressource dar, sondern höchstens einen Teil einer Ressource.[21]

Eine URI besteht aus 5 folgenden Komponenten: Schema, Authority, Path, Query und Fragment (siehe Abbildung 2.1).[22]

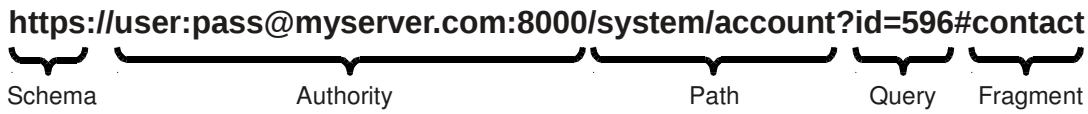

Abbildung 2.1: Aufbau einer URI

Das **Schema** befindet sich am Anfang der URI und ist für die Interpretation der folgenden Komponenten wichtig. Für RESTful Web Services sind nur die URI-Schemata „http“ und „https“ von Bedeutung. Das erste Schema deutet auf eine unverschlüsselte und das zweite Schema auf eine über SSL/TLS verschlüsselte HTTP-Verbindung hin.[23]

Darauf folgt die **Authority**-Komponente mit optionaler Benutzerinformation, einem Hostnamen und ebenfalls optionalem Port. Die optionale Angabe eines Passwortes in der Benutzerinformation ist veraltet und sollte nicht mehr verwendet werden, da diese Informationen im Klartext übertragen werden.[24]

Der **Path** (Pfad) enthält Angaben, die meist hierarchisch organisiert sind. Er identifiziert die Ressourcen vollständig oder teilweise (zusammen mit Query-Komponente), die einzelnen Path-Elemente werden durch Schrägstriche getrennt.[25]

Die optionale **Query**-Komponente (Abfrage-Komponente) enthält Angaben, die nicht hierarchisch organisiert sind. Sie identifiziert die Ressourcen zusammen mit der Path-Komponente und wird mit einem Fragezeichen „?“ eingeleitet.[26]

Gemäß RFC 3986 ist die Auswertung von Query-Inhalt anwendungsabhängig. So werden häufig die „Schlüssel=Wert“-Paare zur Identifizierung von Ressourcen verwendet.

[21] Vgl. [45] S. 81.
[22] Vgl. [35] Kapitel 3. Syntax Components.
[23] Vgl. [53] S. 38.
[24] Vgl. [35] Kapitel 3.2. Authority.
[25] Vgl. [35] Kapitel 3.3. Path.
[26] Vgl. [35] Kapitel 3.4. Query

Die Paare werden durch das Kaufmanns-Und „&“ verkettet.[27]

Als Alternative zur Query-Komponente kann man **Matrixparameter** verwenden. Die Parameter werden durch das Semikolon verkettet. Statt Semikolon kann auch ein Komma für Verkettung von Parametern verwendet werden, falls die Ordnung von übergebenen Werten wichtig ist. Die Verwendung von Matrixparameter bzw. kommagetrennter Parameter kann in folgenden Fällen sinnvoll sein: [28]

- Zwischenspeichern von Ressourcen durch Proxy-Server (die URIs mit Query-Komponente werden durch einige Proxy-Server nicht zwischengespeichert);
- Übergabe von booleschen Werten: *http://example.com/customers;descending*;
- Festlegung einer Untermenge: *http://example.com/users;start=100*;
- Übergabe von Parameter in der Mitte von Hierarchien: *http://example.com/earth/50.59,12.59/Mittweida*.

Weder die „Schlüssel=Wert“-Paare noch die Matrixparameter oder die kommagetrennten Parameter sind in HTTP- oder URI-Standard definiert. Die „Schlüssel=Wert“-Paare sind jedoch weit verbreitet und bei HTML-Formularen standardisiert.[29]

Die ebenfalls optionale **Fragment**-Komponente referenziert die Ressourcenteile und wird immer clientseitig ausgewertet. Ein bekanntes Beispiel für die Verwendung von Fragmenten ist der Einsatz von Anker-Elementen in HTML.[30]

2.4.1 Absolute URI und relative Referenz

Eine absolute URI besteht aus den URI-Komponenten, wobei die Komponente Schema, Authority oder/und Path Pflichtbestandteile sind. Die Komponente Query ist optional. Die Beispiele sind: [31]

- *http://example.com/*,
- *http://example.com/user/12*,
- *http://example.com/orders?year=2010&month=jan*.

Neben einer absoluten URI ist die relative Referenz im Web weit verbreitet. Man erkennt eine Referenz daran, dass sie nicht mit einem Schema beginnt. Eine relative Referenz besteht aus den Komponenten Authority, Path, Query und Fragment. Die Komponenten

[27] Vgl. [35] Kapitel 3.4. Query und [53] S. 40.
[28] Vgl. [53] S. 40, 44 und [45] S. 117–119.
[29] Vgl. [53] S. 40 und [45] S. 119.
[30] Vgl. [35] 3.5. Fragment und [53] S. 41.
[31] Vgl. [35] 4.3. Absolute URI.

Authority, Query und Fragment sind optional. Eine relative Referenz bezieht sich immer auf eine Base-URI.[32]

Die Beispiele sind (Base-URI ist *http://example.com/sys/order/*):

- *./12* ⇒ *http://example.com/sys/order/12*,
- */user/52* ⇒ *http://example.com/user/52*,
- *../orders?year=2010* ⇒ *http://example.com/sys/orders?year=2010*,
- *//example.org/order/45* ⇒ *http://example.org/order/45*.

2.4.2 URI-Design

Obwohl das URI-Design beim Entwurf von RESTful Anwendungen nicht von höchster Bedeutung ist, kann es unter anderem sinnvoll sein über URI-Schemata nachzudenken. Elegante und logische URIs erleichtern die Arbeit für Entwickler und verbessern die Bedienbarkeit für Endanwender.[33]

Für URI-Entwurf sollte man Substantive und keine Verben verwenden. Nicht umsonst werden die RESTful Architekturen als ROA bezeichnet. Die Verwendung von Verben deutet meistens auf eine RPC-orientierte Anwendung hin.[34]

Die Verwendung von Hierarchien stellt eine wichtige und nachvollziehbare Regel dar. Die hierarchischen Beziehungen sollen sich entsprechend in den URI-Pfaden widerspiegeln. Dabei soll das Vortäuschen einer Hierarchie vermieden und evtl. auf Query- bzw. Matrixparameter ausgewichen werden.[35]

Anschließend sollen noch einige URI-Grenzfälle betrachtet werden. Auf eine Ressource können durchaus mehrere URIs zeigen. So können z.B. die URIs */bericht/2004/q3* und */bericht/2004q3* auf eine Ressource zeigen. Dagegen zeigt jede URI genau auf eine Ressource, sonst wäre es keine eindeutige Identifikation.[36]

Die URIs */bericht/2011* und */bericht/aktuell* verweisen andererseits nicht auf die gleiche Ressource, sondern nur auf die gleichen Daten. Die Idee hinter diesen URIs unterscheidet sich stark: Während die erste URI einen Bericht aus dem Jahre 2011 adressiert, so verweist die zweite URI auf einen stets aktuellen Bericht. Somit sind es zwei unterschiedliche Ressourcen.[37]

[32] Vgl. [35] 4.2. Relative Reference.
[33] Vgl. [53] S. 42.
[34] Vgl. [53] S. 42–43.
[35] Vgl. [53] S. 43 und [45] S. 118–119.
[36] Vgl. [45] S. 83.
[37] Vgl. [45] S. 83.

2.5 Repräsentationen

Die Repräsentationen sind Darstellungen einer Ressource. Die Ressourcen müssen mindestens eine Repräsentation aufweisen. Bei mehreren Repräsentationen ist zu beachten, dass alle Repräsentationen gleichermaßen gültig sind.[38]

Allgemein lässt sich jede Ressource im REST auch als eine Idee betrachten: Die Idee legt dabei fest, wie die Informationen beim Anwendungsentwurf aufgeteilt werden. Die Ressourcen stellen somit keine Daten, sondern eine Entwickler-Idee dar. In diesem Sinne lässt sich eine Ressource über ein Netzwerk nicht übertragen. Die Idee muss zuerst auf die Daten abgebildet werden, um später als eine Repräsentation übertragen werden zu können.[39]

Weiterhin kann die Möglichkeit mehrere Repräsentationen für eine Ressource anzubieten den potenziellen Benutzerkreis erheblich erweitern. Man kann mit mehreren Repräsentationen viele unterschiedliche Formate anbieten und somit die Interoperabilität stark verbessern.

Es wird zwischen standardisierten und benutzerdefinierten Formaten unterschieden. Dabei ist ein standardisiertes Format einem benutzerdefinierten Format nach Möglichkeit vorzuziehen. Ein standardisiertes Format verstehen ohne weiteres mehr potenzielle Clients, obwohl es meistens nicht optimal zu den Anforderungen einer spezieller Anwendung passt.[40]

Die meistverbreiteten Repräsentationsformate sind XHTML, XML und JSON. Wobei Mikroformate wie hCard, hCalendar, Nofollow etc. in letzter Zeit eindeutig an Bedeutung gewonnen haben. Sie erweitern XHTML mit semantischen Annotationen, welche leichter aus Webseiten extrahiert werden können. Dies ermöglicht den Einsatz von identischen Formaten für den Menschen und die Maschine.[41]

Einen ähnlichen Ansatz verfolgt das System RDF. Es dient zur Beschreibung von Ressourcen, ist jedoch weitaus komplexer und mächtiger als die Mikroformate.[42]

Das gewünschte Repräsentationsformat wird in dem HTTP-Accept-Header festgelegt. Weiterhin kann die Angabe des Formates auch in der URI erfolgen, was den menschlichen Benutzern sehr entgegen kommt, sodass der Webbrowser alle zur Verfügung stehende Repräsentationen aufrufen kann.[43]

[38] Vgl. [53] S. 31.
[39] Vgl. [45] S. 81.
[40] Vgl. [53] S. 82.
[41] Vgl. [45] S. 259–263.
[42] Vgl. [45] S. 266–267.
[43] Vgl. [45] S. 390.

2.6 Verben

Die Ressourcen stellen die Objekte dar, mit denen man in der Lage sein soll zu interagieren. Diese Aufgabe übernehmen die Verben, welche im Internet durch HTTP-Methoden repräsentiert werden.[44]

Die HTTP-Methoden sind in RFC 2616 definiert: [45]

- **GET** – ruft die Information ab, die durch die Request-URI identifiziert ist;
- **HEAD** – ruft die Meta-Information ab, die durch die Request-URI identifiziert ist;
- **POST** – erstellt eine neue Subressource;
- **PUT** – erstellt bzw. aktualisiert eine Ressource, die durch die Request-URI identifiziert ist;
- **DELETE** – löscht eine Ressource, die durch die Request-URI identifiziert ist;
- **OPTIONS** – ermittelt die Fähigkeiten und Kommunikationsmöglichkeiten eines Servers;
- **TRACE** – liefert eine Loop-Back-Anfrage zurück;
- **CONNECT** – stellt einen SSL-Tunnel bei einem Proxyserver dynamisch zur Verfügung.

RESTful Web Services verwenden hauptsächlich nur die HTTP-Methoden GET, POST, PUT, DELETE, selten auch HEAD und OPTIONS.[46] Man unterscheidet zwischen sicheren und idempotenten Methoden: [47]

- **sichere Methoden** dürfen auf dem Server nichts verändern (dadurch sollen Seiteneffekte bei den Ressourcen vermieden werden);
- **idempotente Methoden** sollen auch bei mehrfacher Ausführung den gleichen Seiteneffekt haben (allerdings ist es möglich, dass eine Abfolge von mehreren Anfragen nicht idempotent ist, auch wenn alle ausgeführte Methoden idempotent sind).

Die folgende Tabelle 2.1 veranschaulicht die Eigenschaften der HTTP-Methoden, die durch RFC 2616 vorgegeben sind. Die ersten zwei Spalten geben an, ob eine Methode sicher und idempotent ist. Die nächsten beiden Spalten geben an, ob eine Ressource dabei durch eine URI identifiziert und gecacht werden kann bzw. darf.[48] Die letzte Spalte gibt an, bei welcher Methode eine Anwendungsinfrastruktur die Semantik und die Bedeutung erkennen und nachvollziehen kann.[49]

[44] Vgl. [57] S. 11.
[45] Vgl. [32] Kapitel 9 Method Definitions.
[46] Vgl. [45] S. 97.
[47] Vgl. [32] Kapitel 9.1 Safe and Idempotent Methods.
[48] Die OPTIONS-Methode kann laut [32] RFC 2616 Kapitel 9.2 OPTIONS nicht gecacht werden. Laut Stefan Tilkov [53] S. 54 ist es allerdings möglich, falls es explizit angezeigt wird.
[49] Vgl. [53] S. 54.

Methode	sicher	idempotent	identifizierbare Ressource	Cache-fähig	sichtbare Semantik
GET	X	X	X	X	X
HEAD	X	X	X	X	X
PUT		X	X		X
POST					
OPTIONS	X	X		O	X
DELETE		X	X		X

Tabelle 2.1: HTTP-Methoden und ihre Eigenschaften, Quelle: [53] S. 54.

2.6.1 GET und HEAD

Die wichtigste und meist verbreitete HTTP-Methode ist GET. Sie ruft die Repräsentationen der Ressourcen ab, die durch die URI identifiziert sind. Die HEAD-Methode ist der GET-Methode sehr ähnlich, sie liefert jedoch nur Meta-Information (HTTP-Header) und keine Repräsentationen (HTTP-Body) zurück. Beide Methoden sind als sicher und idempotent definiert.[50]

Die GET-Methode ist sehr vielfältig. So kann eine bedingte GET-Methode (conditional GET) erstellt werden, indem man in dem Anfrage-Header die Felder „If-Modified-Since“, „If-Unmodified-Since“, „If-Match“, „If-None-Match“ oder „If-Range“ definiert. Das „If-Modified-Since“-Feld teilt z. B. mit, dass eine Repräsentation zurückgeliefert werden soll, falls die Ressource seit dem definierten Datum geändert wurde.[51]

Falls die GET- oder HEAD-Anfrage ordnungsgemäß verläuft, wird der HTTP-Ergebniscode „200 OK“ mit der Repräsentation bzw. mit der Meta-Information einer Ressource zurückgeliefert. Bei einer „If-Modified-Since“-GET-Anfrage kann HTTP-Ergebniscode „304 Not Modified“ zurückgeliefert werden, falls die Ressource seit dem definierten Datum nicht aktualisiert wurde.[52]

Obwohl die GET-Methode als sicher und idempotent definiert ist, lässt sich das Verhalten in einer eigenen Applikation fast beliebig anpassen, sodass die Semantik nicht mehr erkennbar ist. Daraus können sich weitreichende Nachteile ergeben, welche die Vorteile des REST-Architekturstils zunichtemachen.[53]

[50] Vgl. [53] S. 49–52.
[51] Vgl. [32] Kapitel 9.3 GET.
[52] Vgl. [32] Kapitel 10.2.1 „200 OK“ und Kapitel 10.3.5 „304 Not Modified“.
[53] Vgl. [53] S. 50.

2.6.2 DELETE und OPTIONS

Die DELETE-Methode ist nicht als sicher, aber als idempotent definiert. Sie ist für das Löschen einer Ressource zuständig. Wird der HTTP-Ergebniscode „200 OK“, „204 No Content“ oder „204 Accepted“ zurückgeliefert, so wurde oder wird die Ressource gelöscht.[54]

Die OPTIONS-Methode liefert die Meta-Information über eine Ressource zurück. Dabei wird unter anderem das Allow-Feld mit den Methoden übermittelt, die eine Ressource unterstützt. Die OPTIONS-Methode ist dementsprechend als sicher und idempotent definiert. Die meisten Server und Frameworks unterstützen diese Methode nicht. Falls die OPTIONS-Anfrage ordnungsgemäß verläuft, wird der HTTP-Ergebniscode „200 OK“ mit der Meta-Information einer Ressource zurückgeliefert.[55]

2.6.3 PUT

Die PUT-Methode erstellt bzw. aktualisiert eine bestehende Ressource. Die Ressource ist durch die Request-URI vorgegeben. Diese Methode ist idempotent, aber nicht sicher. Kommt z. B. nach einer PUT-Anfrage keine Antwort zurück, so kann die Anfrage im Zweifelsfall mehrmals wiederholt werden.[56]

Die PUT-Anfrage übermittelt die Repräsentation einer Ressource zu dem Server. Der Server soll dabei das Format der übermittelten Repräsentation unterstützen und über das vorliegende Format informiert sein (über URI-Endung bzw. HTTP-Content-Type-Header). Die Informationen werden sinngemäß übernommen, das heißt, dass der Server die Daten ändern, ignorieren oder ergänzen darf .[57]

Falls die PUT-Änderungsanfrage ordnungsgemäß verläuft und eine vorhandene Ressource aktualisiert wird, wird der HTTP-Ergebniscode „200 OK“ mit einer Repräsentation oder „204 No Content“ ohne Repräsentation zurückgeliefert. Wird der HTTP-Ergebniscode „201 Created“ oder „202 Accepted“ zurückgeliefert, so wurde oder wird eine neue Ressource erstellt.[58]

2.6.4 POST

Die POST-Methode ist weder als sicher und noch als idempotent definiert, Sie hat im Wesentlichen zwei Einsatzfelder: [59]

[54] Vgl. [53] S. 53 und [32] Kapitel 9.7 DELETE.
[55] Vgl. [32] Kapitel 9.2 OPTIONS.
[56] Vgl. [53] S. 52.
[57] Vgl. [53] S. 52.
[58] Vgl. [32] Kapitel 9.6 PUT und Kapitel 10.2.2 „201 Created“.
[59] Vgl. [45] S. 99.

- REST-konformes Anlegen einer neuen Ressource,
- eine an den RPC-Stil angelehnte Verwendung, wobei die Methode überladen wird; dadurch kann eine beliebige Verarbeitung angestoßen werden.

Die Methode POST kann genauso wie PUT für das Anlegen einer neuen Ressource verwendet werden. Dabei wird im Gegensatz zu der PUT-Methode eine untergeordnete Ressource erstellt, welche nur in Bezug auf eine andere „Eltern“-Ressource existiert. Die Anfrage enthält nur die URI einer „Eltern“-Ressource und keine Ressource-URI, da diese unbekannt ist. Die Ressource-URI wird erst durch den Server bestimmt und dem Client mitgeteilt.[60]

Der wichtigste Unterschied zwischen PUT und POST ist folgender: [61]

- die PUT-Methode wird verwendet, falls der Client die neue URI kennt bzw. selbst generieren kann;
- die POST-Methode wird verwendet, falls der Client die neue URI nicht kennt und nicht generieren kann; der Server ist dann für die Generierung einer neuen URI zuständig.

Das zweite Einsatzfeld von POST ist sehr allgemein und kommt aus dem RPC-Stil hervor. Die POST-Methode lässt sich missbrauchen, um beliebige Funktionalitäten anzustoßen. Die Funktionalität wird dabei in den übermittelten Daten codiert, was gegen REST-Prinzipien verstoßt. Trotzdem bleibt dieser Ansatz in der Praxis oft als letzter Ausweg, so z. B. das Tunneln von beliebigen Operationen, die durch andere Methoden nicht bzw. nicht mit ähnlichem Aufwand zu realisieren sind.[62]

Genau wie bei der PUT-Methode werden beim Anlegen einer neuen Ressource der HTTP-Ergebniscode „201 Created“ oder „202 Accepted“ zurückgeliefert. In anderen Fällen wird der HTTP-Ergebniscode „200 OK“ mit einer Repräsentation oder „204 No Content“ ohne Repräsentation zurückgeliefert.[63]

2.6.5 Unterstützung von HTML-Formularen

Häufig sind die Ressourcen sowohl über den Webbrowser als auch über eine Applikation-Schnittstelle verfügbar. Die HTML-Formulare unterstützen aber nur die Methoden GET und POST, nicht aber PUT, DELETE, HEAD und OPTIONS. In der Zukunft können die Methoden PUT und DELETE durch HTML 5 unterstützt werden, das Problem mit HEAD und OPTIONS bleibt anscheinend unverändert.[64]

[60] Vgl. [53] S. 52–53 und [45] S. 99.
[61] Vgl. [45] S. 99.
[62] Vgl. [53] S. 52-53 und [45] S. 101–102.
[63] Vgl. [32] Kapitel 9.5 POST und Kapitel 10.2.2 „201 Created“.
[64] Siehe [2] Supporting PUT and DELETE with HTML FORMS.

Da die HTML-Formulare die POST-Methode unterstützen, kann eine Art Tunnel für alle anderen nicht unterstützten Methoden hergestellt werden, indem ein zusätzliches Feld in das Formular aufgenommen wird. Der Webbrowser übermittelt dann die Daten immer noch mithilfe der POST-Methode. Auf der Serverseite lässt sich eine Umleitung einrichten, die solche Anfragen direkt an die Funktion, die auch andere PUT-Anfragen bearbeitet, weiterleitet. Diese Lösung wird unter anderem auch von Ruby on Rails eingesetzt.[65]

2.7 Hypermedia

Das Konzept der Hypermedia lässt sich beim Surfen im Web gut nachvollziehen. Beim Navigieren zwischen den Seiten wird der Zustand einer Anwendung durch das Klicken auf die Hyperlinks oder durch das Ausfüllen von Formularen stets geändert. Hypermedia ist ein fester Bestandteil der menschlichen Online-Aktivitäten und wird nun auch zur Interaktion zwischen den Anwendungen eingesetzt.[66]

Hypermedia stellt die zentrale Idee der Steuerung des Anwendungszustands beim REST-Architekturstil dar. Die Verknüpfungen zwischen Ressourcen erlauben die Steuerung und die Kommunikation zwischen verschiedenen Ressourcen. Somit kann eine Anwendung auch ohne Kenntnis der Struktur eines Services an die nötige Ressource gelangen und den Ressourcenstatus ändern.[67]

Es gibt viele für Hypermedia geeignete Formate, diese können sich in der Realisierung leicht unterscheiden. Das bekannteste Format ist XHTML, welches durch Hyperlinks die Informationen verknüpft.[68] Die Implementierungen in den Formaten XML und JSON sehen dem XHTML-Format ähnlich: [69]

- XHTML:

```
<a rel="self" href="http://www.example.org/person/john" />
```

- XML:

```
<link rel="self" href="http://www.example.org/person/john"
/>
```

- JSON:

```
"link" : {
    "rel" : "self",
    "href" : "http://www.example.org/person/john"
}
```

[65] Vgl. [53] S. 55–56.
[66] Vgl. [57] S. 93.
[67] Vgl. [53] S. 10–11 und [45] S. 94.
[68] Vgl. [57] S. 97.
[69] Vgl. [1] S. 56–59.

3 Ruby on Rails

Dieses Kapitel befasst sich mit der Programmiersprache Ruby und dem Web Framework Ruby on Rails. Es werden vor allem wichtige Aspekte betrachtet, die für die Arbeit mit Ruby on Rails von Bedeutung sind.[70]

Die Programmiersprache Ruby wurde im Jahr 1993 von Yukihiro Matsumuto entworfen und erstmals im Jahr 1995 veröffentlicht.[71] Ruby ist eine dynamische Programmiersprache mit einer komplexen und ausdrucksstarken Grammatik, welche auch für C- und Java-Programmierer leicht zu erlernen ist. Die Programmiersprache selbst wurde von Lisp, Smalltalk und Perl inspiriert. Sie ist eine rein objektorientierte Sprache, eignet sich aber auch für prozedurale und funktionale Programmierstile.[72]

Ruby on Rails, kurz Rails oder RoR, ist ein in der Programmiersprache Ruby geschriebenes Web Framework. Ruby on Rails wurde von David Heinemeier Hansson entwickelt und im Jahr 2004 unter MIT-Lizenz als Open Source veröffentlicht. Rails wurde dabei aus einer bestehenden Webanwendung namens Basecamp extrahiert.[73] Seitdem hat sich Ruby on Rails zu einem mächtigen und bekannten Framework für die Entwicklung dynamischer Webanwendungen geworden. Rails hat mehrmals bewiesen, dass es sich schnell an neue Entwicklungen und Trends in der Web-Technologie anpassen kann. So hat Ruby on Rails als erstes Framework den REST-Architekturstil beinah vollständig umgesetzt und bietet eine besonders gute Unterstützung zum Erstellen von RESTful Webanwendungen.[74]

3.1 Ruby-Grundlagen

Ruby ist eine interpretierte Programmiersprache, die für schnelle und einfache objektorientierte Programmierung entwickelt wurde. Die Programme werden zur Laufzeit von einem Ruby-Interpreter analysiert und ausgeführt, daraus folgt die relativ geringe Ausführungsgeschwindigkeit.[75]

Ruby ist eine vollständig objektorientierte Sprache. In Ruby ist alles ein Objekt und folgt dem Prinzip von Klassen, Methoden und Vererbung. Die Funktionalitäten wie Singleton-Methoden und Mixins-Module werden direkt unterstützt.[76]

[70] Als weiterführende Literatur sind die Werke „Programming Ruby 1.9“ [51], „The Ruby Programming Language“ [17], „Ruby on Rails 3 Tutorial“ [18] und „Rails 3 in Action“ [21] empfehlenswert.
[71] Vgl. [21] S. 2–5.
[72] Vgl. [17] Kapitel 1. Introduction.
[73] Vgl. [21] S. 2–5.
[74] Vgl. [18] S. 3–4 und [54].
[75] Vgl. [49] „What is ruby?“ und [31] Kapitel 4 Einführung in Ruby.
[76] Vgl. [49] „What is ruby?“ und [17] Kapitel 7. Classes and Modules.

Ruby ist eine dynamisch typisierte Sprache. Die Variablen sind nicht typisiert und die Variablendeklarationen sind unnötig. Außerdem führt Ruby eine automatische Speicherverwaltung durch und bietet ein API zur Reflexion und Metaprogrammierung an. Man kann somit nicht nur die Methoden zur Laufzeit definieren bzw. ändern, sondern auch den Zustand und die Struktur des Programms vom Programm selbst verändern lassen.[77]

3.1.1 OOP in Ruby

Die Definition einer Klasse beginnt mit dem Schlüsselwort *class* und endet mit dem Wort *end*, wie die meisten Ruby-Konstrukte. Der Name einer Klasse soll immer mit einem Großbuchstaben beginnen. Bei Klassen und Modulen gilt die CamelCase-Konvention.[78]

Wie das Listing 3.1 zeigt, werden die Semikolons am Ende der Anweisungen nicht gebraucht, solange jede Anweisung auf einer neuen Zeile beginnt. Einzeilige Kommentare beginnen mit einem Routen-Zeichen #. Die Einrückungen sind nicht signifikant, trotzdem werden Zwei-Leerzeichen-Einrückungen für bessere Lesbarkeit empfohlen.[79]

Jedes Objekt ist eingekapselt, somit ist ein Zugriff auf die Objekteigenschaften nur mit den Methoden des entsprechenden Objektes möglich. Für diesen Zweck werden die Getter- und Setter-Methoden definiert, siehe Zeilen 3 bis 9 in Listing 3.1.[80]

Die Definition einer Methode beginnt mit dem Schlüsselwort *def*. Bei der Namensgebung von Methoden und Variablen gilt die Unterstrich-Konvention: Der Name soll mit einem Kleinbuchstaben bzw. mit einem Unterstrich beginnen und die einzelnen Wörter sollen durch einen Unterstrich getrennt sein. Die Klammern sind in den meisten Fällen optional. Als Rückgabe wird der Wert des zuletzt ausgewerteten Ausdrucks zurückgegeben, somit wird oftmals auf eine *return*-Anweisung verzichtet, siehe Zeile 8 in Listing 3.1.[81]

Da die Kombination von Getter- und Setter-Methoden sehr häufig auftritt, bietet Ruby vorgefertigte automatisierte Zugriffsmethoden an: *attr_reader* und *attr_accessor*. Beide Methoden erwarten eine beliebige Anzahl von Attributnamen als Symbole. Die Methode *attr_reader* erstellt eine triviale Getter-Methode, die Methode *attr_accessor* erstellt triviale Getter- und Setter-Methoden, siehe Zeile 11 in Listing 3.1.[82]

[77] Vgl. [49] „What is ruby?“ und [17] Kapitel 8. Reflection and Metaprogramming.
[78] Vgl. [17] Kapitel 7.1.1. Creating the Class.
[79] Vgl. [51] S. 38.
[80] Vgl. [17] Kapitel 7.1.5. Accessors and Attributes.
[81] Vgl. [51] S. 38–40.
[82] Vgl. [17] Kapitel 7.1.1. Creating the Class und [58] Kapitel 2.7.1. Getter und Setter.

```ruby
class Point                 # Beginn der Klassendefinition

  def x=(x)                 # Setter-Methode fuer x
    @x = x
  end

  def x                     # Getter-Methode fuer x
    @x
  end

  attr_accessor :y          # Zugriffsmethoden fuer y generieren

  def initialize(x,y)       # Initialisiert ein neues Objekt
    @x, @y = x, y
  end

  def to_s                  # Gibt ein Objekt als Zeichenkette
    "(#@x,#@y)"             # zurueck
  end

end                         # Ende der Klassendefinition
```

Listing 3.1: Point-Klasse, Quelle: angelehnt an [17] Kapitel 7. Classes and Modules.

Beim Erstellen von neuen Instanzen mit der Methode *new* wird die private Methode *initialize* automatisch aufgerufen. Alle Argumente der Methode *new* werden dabei an die Methode *initialize* weitergegeben. Diese Methode ist eine Art Konstruktor und hat die Aufgabe eine Instanz in einen definierten Anfangszustand zu überführen, indem sie z. B. die Attribute initialisiert.[83] Jede Klasse sollte außerdem eine *to_s*-Methode beinhalten. Diese Methode gibt eine Zeichenkette zurück, welche das Objekt repräsentiert.[84]

Es wird zwischen 4 Variablenarten unterschieden: [85]

- lokale Variable ist in dem Block gültig, in dem sie definiert wurde;
- Instanzvariable ist in einer Instanz gültig;
- Klassenvariable ist in einer Klasse gültig;
- globale Variable ist überall im Programm gültig.

Die Instanzvariablen beginnen mit einem At-Zeichen @ und die Klassenvariablen beginnen mit zwei At-Zeichen @@. Die globalen Variablen fangen mit Dollarzeichen $ an, die lokalen Variablen haben kein Prefix.[86]

[83] Vgl. [17] Kapitel 7.1.3. Initializing a Point.
[84] Vgl. [17] Kapitel 7.1.4. Defining a to_s Method.
[85] Vgl. [51] S. 334–335 und [58] Kapitel 2.6. Variablen.
[86] Vgl. [51] S. 334–335.

3.1.2 Kontroll- und Datenstrukturen

Ähnlich wie viele andere Sprachen besitzt Ruby die üblichen Kontrollstrukturen wie *if*, *while*, *for*, *case* u. ä. Die Kontrollstrukturen wie *if*, *unless*, *while* und *until* können als Anweisungsmodifikatoren benutzt werden, um die Bedingungsanweisung am Ende einer normalen Anweisung zu setzen. Auf das Klammern von logischen Ausdrucken wird in den meisten Fällen verzichtet.[87]

Als vordefinierte Datenstrukturen stehen in Ruby Arrays und Hashes zur Verfügung. Die Elemente in einem Array sind geordnet und können nur mit Zahlen indiziert werden. Die Elemente in einem Hash sind auch geordnet und können mit beliebigen Objekten indiziert werden. Beide Datenstrukturen sind nicht typisiert, aber dynamisch wachsend. Arrays sind somit effizienter als Hashes, bieten aber weniger Flexibilität.[88]

3.1.3 Blöcke und Iterationen

Blöcke und Iterationen stellen die entscheidenden Features von Ruby dar. Alle nicht trivialen Ruby-Programme weisen eine Blockstruktur auf, die Blöcke sind durch Zeichensetzung oder durch Schlüsselwörter wie *do* und *end* getrennt. Von besonderer Bedeutung sind die Blöcke in der Verbindung mit Iteratormethoden.[89]

Wird ein Block mit einer Methode verbunden, so kann die Methode den Block beliebig oft aufrufen, indem eine *yield*-Anweisung durch den Block ersetzt wird. Dieses Vorgehen kann man mit einer zusätzlichen Parameterübergabe vergleichen, mit dem Unterschied, dass dabei Anweisungen und keine Parameter übergeben werden: [90]

```
1  1.upto(5) { |x| puts x*x } # Quadrate fuer Zahlen 1..5
2                             # Ausgabe: 1 4 9 16 25
3  a = [ 1, 3, "cat" ]        # Array definieren
4  a.each do |x|              # Element duplizieren
5    puts x*2                 # Ausgabe: 2 6 catcat
6  end
7
8  def verdreifach            # Definiere Funktion, die jeden
9    yield; yield; yield      # Anweisungsblock 3-fach ausfuehrt
10 end
11
12 verdreifach {puts "Ho!␣"}  # Ausgabe: Ho! Ho! Ho!
```

Listing 3.2: Blöcke und Iterationen, Quelle: angelehnt an [17] Kapitel 5.4. Blocks und [17] S. 77.

[87] Vgl. [51] S. 43–44.
[88] Vgl. [17] Kapitel 3.3. Arrays und 3.4. Hashes.
[89] Vgl. [51] S. 43–44 und [17] 2.2.1. Block Structure in Ruby.
[90] Vgl. [51] S. 75–80 und [17] 5.4. Blocks.

3.2 Rails-Grundlagen

Die Rails-Philosophie basiert auf zwei einfachen Konzepten. Das erste Konzept heißt DRY („Don't Repeat Yourself" – „Wiederhole dich nicht"). Durch Anwendung des DRY-Prinzips versucht Rails die Redundanz zu vermeiden, in dem es z. B. an vielen Stellen Quellcode automatisch generiert. Das zweite Konzept heißt „Convention over Configuration". Das reduziert die Komplexität der Anwendungskonfiguration erheblich, solange die Entwickler sich an die üblichen Rails-Konventionen halten.[91]

Ein weiteres wichtiges Prinzip der Entwicklung von Rails-Anwendungen ist TDD („Test-Driven Development" oder „testgesteuerte Programmierung"), dies ermöglicht eine relativ fehlerfreie Implementierung der Software mit hoher Komplexität. Auch das Model-View-Controller-Entwurfsmuster hat Rails stark beeinflusst, sodass die Entwicklung von Ruby-Anwendungen meistens in drei voneinander unabhängigen Schichten stattfindet.[92]

Rails gehört außerdem zu den wenigen Frameworks, die den REST-Architekturstil nicht nur gut unterstützen, sondern auch konsequent umsetzen. Bereits ab der Version 2.0 setzt Rails die REST-Prinzipien standardmäßig um.[93]

3.2.1 Projektverzeichnisstruktur und Arbeitsumgebung

Beim Anlegen eines neuen Projektes wird eine feste Verzeichnisstruktur incl. aller Konfigurationsdateien angelegt, siehe Tabelle 3.1. Die gesamte Organisation des Projektes wird somit durch Rails übernommen, was die Wartbarkeit von Rails-Anwendungen unterstützt und die Einarbeitungszeit stark verkürzt.[94]

Rails kennt drei unterschiedliche Umgebungen, die Environments genannt werden: [95]

- **Development** – wird bei Entwicklung eingesetzt, alle Fehler werden geloggt und die Debug-Information wird zur Auswertung angeboten;
- **Testing** – wird bei Entwicklung von eigenen Tests eingesetzt, die Test-Fehler können mitgeloggt werden;
- **Production** – wird beim produktiven Einsatz verwendet, die Debug- und Testinformationen werden hier nicht angeboten.

[91] Vgl. [31] Kapitel 1.1 Wie entstand Rails und [37] Kapitel 1 Einführung.
[92] Vgl. [31] Kapitel 6.1 Was ist TDD und [18] 3.2 Our First Tests.
[93] Vgl. [54].
[94] Vgl. [31] Kapitel 1.1 Wie entstand Rails.
[95] Vgl. [10] S. 162.

Datei/Verzeichnis	Beschreibung
Gemfile	In dieser Datei wird angegeben, welche Abhängigkeiten bei dem Projekt benötigt werden.
README	Diese Datei enthält eine kurze Anleitung mit wichtiger Information für Anwender.
Rakefile	Diese Datei sucht und lädt Aufgaben, die von der Kommandozeile ausgeführt werden können. Die Aufgabenstellungen sind über die Komponenten des Rails definiert. Benutzerdefinierte Aufgaben werden als Dateien in das Verzeichnis *lib/tasks* hinzugefügt.
app/	Alle Model-, Controller-, View- und Helper-Dateien befinden sich in diesem Verzeichnis.
config/	Hier werden alle Konfigurationsdateien für die Rails-Umgebung gespeichert.
config.ru	Diese Datei enthält Rack-Konfigurationen, die für das Starten der Rails-Anwendung durch einen Rack-basierten Server gebraucht werden.
db/	Alle Schema- und Migration-Dateien für die Datenbank sind in diesem Verzeichnis enthalten.
doc/	Hier wird die Dokumentation abgelegt.
lib/	Dieses Verzeichnis enthält Erweiterungsmodule, die zu keiner Anwendungsschicht direkt gehören.
log/	Die Log-Dateien werden in diesem Verzeichnis gespeichert.
public/	Dieses Verzeichnis enthält die statischen Dateien, welche direkt von außen aufrufbar sind.
script/	Dieses Verzeichnis enthält Skripte, welche die Anwendung starten. Hier werden auch die benutzerdefinierten Skripte abgelegt, welche zur Ausführung bzw. Bereitstellung der Anwendung notwendig sind.
test/	Hier werden die Testdateien abgelegt.
tmp/	Das ist Verzeichnis für temporäre Dateien.
vendor/	In diesem Verzeichnis befindet sich der Quellcode von Drittanbietern.

Tabelle 3.1: Projektverzeichnisstruktur einer Rails-Anwendung, übersetzt und verkürzt aus [42] Getting Started with Rails: 3.2 Creating the Blog Application.

Hauptsächlich wird in der Umgebung Development gearbeitet und beim Durchführen von Tests in die Umgebung Testing gewechselt. Die Umgebung Development bringt einen eingebauten und konfigurierten WEBrick-Server und eine SQLite-Datenbank mit, was den Entwicklungsanfang erleichtert. Für die Umgebung Production sollte man aufgrund der geringen Performance auf einen anderen Server und auf eine andere Datenbank ausweichen, z.B. auf Apache und MySQL.[96]

3.2.2 Model-View-Controller

Ruby on Rails fördert den Einsatz des MVC-Entwurfsmuster, somit wird die Entwicklungszeit verringert, die Erweiterbarkeit und die Wartbarkeit der Anwendung verbessert. Eine Rails-Anwendung besteht meistens aus drei relativ unabhängigen Schichten: [97]

- **Model** repräsentiert die Geschäftslogik einer Anwendung, welches alle Funktionalitäten zur Erfüllung des eigentlichen Zweckes zur Verfügung stellt;
- **View** kümmert sich um die Präsentationslogik, die aus der Model-Schicht stammende Daten darstellt;
- **Controller** stellt die Programmsteuerungslogik dar, welche Model- und View-Schichte koordiniert und steuert.

Die strikte Implementierung des MVC-Entwurfsmusters in Rails spiegelt sich auch in der Architektur des Rails-Frameworks wider. Die Rails-Komponenten werden in einer ähnlichen Weise aufgeteilt, um den Entwicklungsprozess optimal zu unterstützen. Das Zusammenspiel von Rails-Komponenten zeigt die Abbildung 3.1. Die von Rails verfolgte Strategie wird in folgenden 10 Schritten beschrieben: [98]

1. Der Webbrowser/Client sendet eine Anfrage an den Webserver.
2. Der Webserver leitet die Anfrage an die Komponente Action Dispatch.
3. Die Komponente Action Dispatch analysiert die Anfrage, bestimmt den verantwortlichen Action Controller und leitet die Anfrage an diesen Controller weiter.
4. Action Controller fragt die benötigten Daten bei der Komponente Active Model nach.
5. Active Model fragt die Daten bei der Datenbank ab (optional).
6. Active Model gibt die Daten dem Action Controller zurück.
7. Die Komponente Action View greift auf die Daten zu und rendert die Vorlage als HTML/XML/JSON-Repräsentation.
8. Die Repräsentation wird an Action Controller zurückgegeben.
9. Action Controller leitet die Repräsentation an den Webserver weiter.
10. Der Webserver beantwortet die Anfrage mit der Repräsentation.

[96] Vgl. [58] Kapitel 3.1.1. Arbeits-Umgebung (Development).
[97] Vgl. [10] S. 155–157.
[98] Vgl. [18] S. 49–58.

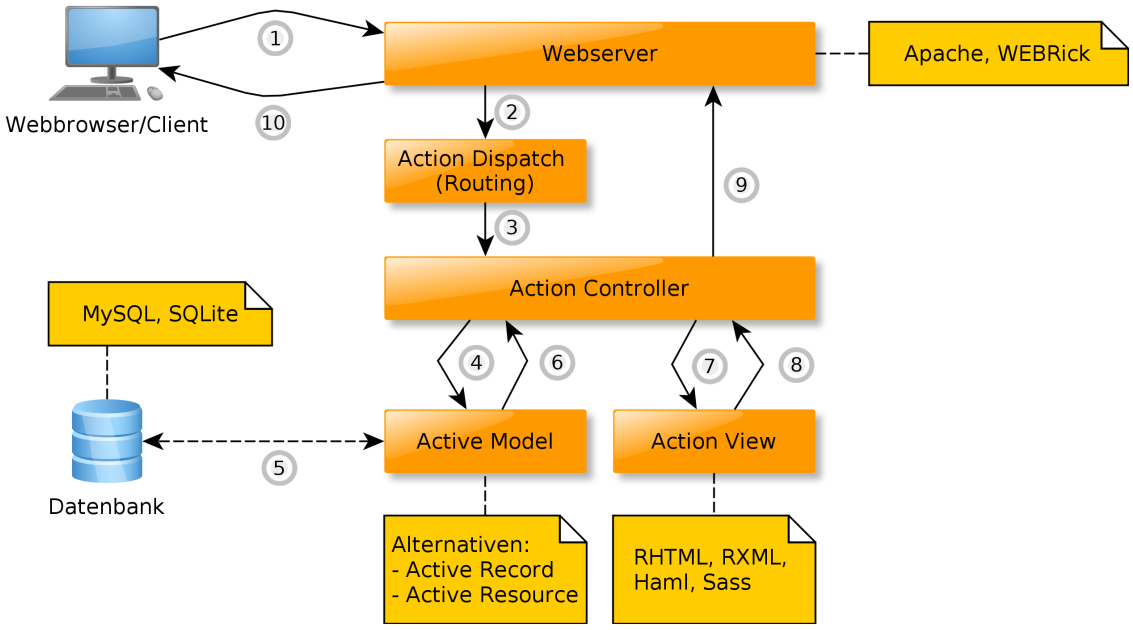

Abbildung 3.1: MVC-Architektur einer Rails-Anwendung. Vergleiche: [18] S. 55.

3.2.3 Active Record

Active Record ist ein Sub-Framework, das unabhängig von Rails eingesetzt werden kann. Es erlaubt den objektorientierten Zugriff auf relationale Datenbanken, was durch objektrelationales Mapping (ORM) erreicht wird. Diese Technologie kann die Effizienz beim Implementieren von datenbankgestützten Anwendungen stark erhöhen, um Time-to-Market so weit wie möglich zu verkürzen. Die DB-Zugriff-Details können dadurch hinten angesetzt werden.[99]

Das Framework ist an keine bestimmte Datenbank gebunden, somit kann eine Anwendung unabhängig von der verwendeten Datenbank entwickelt und eingesetzt werden. Active Record bietet jedoch nicht nur grundlegende CRUD-Funktionalitäten wie Create, Read, Update und Destroy (Fähigkeit einen Datensatz anlegen, lesen, aktualisieren und löschen), sondern auch Datenmigration, Datenvalidierung, Objekt-Callbacks und Suchunterstützung.[100]

3.2.4 Active Resource

Active Resource ist eine Rails-Technologie, welche Ruby-Objekte und RESTful Web Services miteinander verbindet. Somit ist man in der Lage auf die Ressourcen einer anderen REST-Anwendung zuzugreifen. Active Resource verfolgt eine identische Philosophie wie Active Record, sodass sie die externen Ressourcen auch über ein objektre-

[99] Vgl. [31] Kapitel 10 Datenbankzugriff mit ActiveRecord und [42] Getting Started with Rails: 2.2.3 Active Model.

[100] Vgl. [10] S. 157–159 und [42] Getting Started with Rails: 6.4 The Model.

lationales Mapping zur Verfügung stellt.[101]

Die Active-Resource-Base-Klasse wurde außerdem so konzipiert, dass sie sich kaum von einer Active-Record-Base-Klasse in der Benutzung unterscheidet. Active Resource bietet alle Methoden zur Veränderung eines Objekt-Lebenszyklus. Somit kann man beim Finden, Erstellen, Aktualisieren oder Löschen einer Ressource direkt auf die angebotene Funktionalität zugreifen.[102]

Die internen Active-Resource-Klassen sind auf den externen REST-Ressourcen abgebildet, ähnlich wie Active-Record-Klassen auf Datenbanktabellen abgebildet sind. Wird eine Anfrage an eine Active-Resource-Klasse erstellt, so wird zuerst eine REST-Anfrage erzeugt und ausgeführt, anschließend wird das Ergebnis in ein gewöhnliches Ruby-Objekt serialisiert.[103]

Möglich wird dieses Mapping durch die Einhaltung von Code- und Protokoll-basierten Konventionen, die von beiden Kommunikationspartner eingehalten werden müssen. Die Konventionen sind in der Rails-API-Dokumentation [43] Kapitel ActiveResource::Base genau dokumentiert.[104]

3.2.5 Active Model

Active Record und Active Resource besitzen einen sehr großen Funktionsumfang, sind aber für das Definieren von spezifischen Modellen nicht geeignet. Aus diesem Grund wurde in der 3. Version von Rails eine neue Bibliothek namens Active Model aus dem Framework Active Record extrahiert.[105]

Diese Bibliothek bietet eine ähnliche Funktionalität wie die Frameworks Active Record oder Active Resource. Somit lassen sich die Module für Validierung, Callbacks, Serialisierung und Übersetzung unabhängig von oben genannten Frameworks einsetzen, was das DRY-Prinzip widerspiegelt.[106]

Active Model beinhaltet viele weitere nützliche Module, unter anderem solche Module, die bei Interaktion mit Action Pack gebraucht werden. Somit sind alle neu entwickelte ORM-Klassen in der Lage mit Action Pack zu interagieren, soweit sie die Active-Model-Schnittstelle implementiert haben.[107]

[101] Vgl. [31] Kapitel 14.5 Zugriff auf einen Webservice mit ActiveResource, [13] Kapitel 15.4 Active Resource und [43] Kapitel ActiveResource::Base.
[102] Vgl. [12] Kapitel 8. Active Resource.
[103] Vgl. [43] Kapitel ActiveResource::Base.
[104] Vgl. [12] Kapitel 8. Active Resource und [43] Kapitel ActiveResource::Base.
[105] Vgl. [13] S. 561 und [42] Ruby on Rails 3.0 Release Notes: 8 Active Model.
[106] Vgl. [13] S. 561.
[107] Vgl. [42] Ruby on Rails 3.0 Release Notes: 8 Active Model.

3.2.6 Action Pack

Die Komponente Action Pack beschäftigt sich mit der Behandlung von Anfragen und mit der Ausgabe von Ergebnissen. Die Anfragen werden durch das Modul Action Dispatch ausgewertet und an den entsprechenden Action Controller weitergeleitet. Die Ergebnisse werden dann entweder in XML/JSON serialisiert bzw. als HTML zurückgeliefert, wobei die XML/HTML-Repräsentationen meistens durch das Rendering von Action-View-Vorlagen entstehen.[108]

Action Dispatch

Die Komponente Action Dispatch bietet eine Implementierung für das Routing von Anfragen in Rails-Anwendungen. Dabei kann sie nicht nur die HTTP-Anfragen analysieren, sondern auch die Cookies und Sessions verwalten.[109]

Ein wichtiger Teil von Action Dispatch, das Modul Routing, ist für die Interpretation und Auswertung von Anfragen verantwortlich. Dabei werden alle notwendigen Informationen aus der URL und dem HTTP-Header herausgefiltert, um eine passende Action zu bestimmen und auszuführen. Unter einer Action versteht man eine öffentliche Methode einer Controller-Klasse, die von der Action-Controller-Klasse vererbt ist. Wurde eine passende Action gefunden, so wird eine neue Instanz des Controllers, der die Action beinhaltet, erstellt und die Action anschließend ausgeführt.[110]

Das Modul Routing bietet somit die Rewrite-Engine-Funktionalität (ähnlich wie mod_rewrite von Apache), die aber direkt in Ruby implementiert ist. Das Modul funktioniert unabhängig vom darunterliegenden Webserver, was die Routen-Konfiguration vereinfacht, da alle Routen in einer einzigen Datei *config/routes.rb* im Projektverzeichnis definiert werden.[111]

Action Controller

Nach dem das Routing den verantwortlichen Controller gefunden hat, übernimmt dieser die weitere Bearbeitung der Anfrage. Eine von Action Controller geerbte Klasse soll aus mindestens einer Action, die eine Anfrage bearbeiten kann, bestehen. Dabei kann das Ergebnis gleich berechnet werden oder es findet eine Umleitung zu einer anderen Action statt.[112]

[108] Vgl. [47] S. 53 und [41].
[109] Vgl. [43] Kapitel ActionDispatch und [42] Ruby on Rails 3.0 Release Notes: 7.3 Action Dispatch.
[110] Vgl. [43] Kapitel ActionDispatch::Routing und [10] S. 160.
[111] Vgl. [43] Kapitel ActionDispatch::Routing.
[112] Vgl. [43] Kapitel ActionController::Base und [42] Action Controller Overview: 1 What Does a Controller Do.

Für die Bearbeitung der Anfrage werden meistens weitere Daten gebraucht, die aus einem Modell (Active Model/Record/Resource) geladen werden. Nach der Bearbeitung der Anfrage werden die Daten evtl. aktualisiert oder gespeichert und das Ergebnis der Bearbeitung direkt oder mithilfe der Komponente Action View zurückgegeben. Somit übernimmt der Controller die Aufgabe eines Vermittlers zwischen den Komponenten Active Model und Action View.[113]

Action View

Action View ist ein weiteres Sub-Framework von Rails, welches für das Erstellen von Repräsentationen eingesetzt wird. Action View bietet die Module Helper, Renderer und Templates an, welche die Generierung von Repräsentationen unterstützen. Allgemein wird zwischen zwei Vorlagen unterschieden: [114]

- Vorlagen mit Dateierweiterungen *.erb* oder *.rhtml* stellen eine Mischung von Template-System ERB und HTML dar; sie erzeugen die HTML-Repräsentationen;
- Vorlagen mit Dateierweiterungen *.builder* oder *.rxml* verwenden für die Generierung von XML-Repräsentationen die Bibliothek Builder::XmlMarkup.

Es existieren außerdem weitere Erweiterungen und Module zur vereinfachten Generierung von Repräsentationen, wie z. B. Haml, Sass oder JavaScript Helper mit Unterstützung für JS-Frameworks Prototype und jQuery.[115]

3.2.7 Scaffolding

Scaffolding (vom Englischen für „Grundgerüst“) bietet einen schnellen Weg eine datenbankgestützte Rails-Anwendung mittels Meta-Programmierung zu generieren. Scaffolding wird bei der Entwicklung und vor allem im Prototyping eingesetzt.[116]

Beim Scaffolding werden anfangs die Spezifikationen wie Modellnamen und Attribute festgelegt. Daraus werden alle nötigen Routen, Actions, Modelle, Views und Tests erzeugt. Das Scaffolding-Ergebnis ist eine voll funktionsfähige Rails-Anwendung.[117]

[113] Vgl. [42] Action Controller Overview: 1 What Does a Controller Do.
[114] Vgl. [43] Kapitel ActionView::Base und [10] S. 159.
[115] Vgl. [43] Kapitel ActionDispatch, `http://haml-lang.com/` und `http://sass-lang.com/`.
[116] Vgl. [10] S. 161 und [58] Kapitel 5.1 Einleitung.
[117] Vgl. [10] S. 161 und [42] Getting Started with Rails: 5 Getting Up and Running Quickly with Scaffolding.

4 Cluster-Management

Dieses Kapitel bietet einen groben Überblick zu Cluster und Cluster-Managementsoftware an.[118] Anschließend wird die Megware Cluster-Managementsoftware näher betrachtet und analysiert.

Dieses Kapitel basiert auf den Kapiteln 2, 3 und 5 aus dem Praktikumsbericht „Entwicklung einer Qt-basierten grafischen Benutzeroberfläche für die Megware Cluster-Managementsoftware“ [7]. Die früheren Darlegungen und Analysen wurden für diese Arbeit allerdings weitestgehend erweitert und überarbeitet.

4.1 Grundaufbau eines Clusters

Ein Cluster (vom Englischen für „Gruppe“, „Schwarm“ oder „Haufen“) kann sehr unterschiedlich aufgebaut sein, die Cluster-Grundkomponenten sind jedoch immer gleich. So besteht ein Cluster aus einer Ansammlung von mindestens 2 Knoten, auch *Nodes* (engl.) genannt. Die Knoten arbeiten parallel und ihre Prozesse kommunizieren über ein Netzwerk. Siehe Abbildung 4.1.[119]

Die Knoten werden unterschieden in Server-Knoten (engl. *server nodes*) und Rechenknoten (engl. *compute nodes*). Die Server-Knoten sind für Zugriff, Verwaltung, Lastverteilung, etc. zuständig. Die Rechenknoten verrichten die eigentliche Rechenarbeit und werden oftmals einfach als Knoten bezeichnet.[120]

Als Netzwerk kommt meistens ein geswitchtes Hochgeschwindigkeitsnetzwerk mit Stern-Topologie zum Einsatz. Abhängig von dem Einsatzszenario und der Auslastung können andere Netzwerktechnologien besser geeignet sein.[121]

Die Anforderungen sind von dem Einsatzfeld des Clusters und von der Problemstellung abhängig, deshalb werden auch die Cluster-Komponenten fallspezifisch ausgewählt und kombiniert. Allgemein gibt es drei unterschiedliche Einsatzziele für einen Cluster: [122]

- Rechenleistungserhöhung – High Performance Computing (HPC),
- Hochverfügbarkeit – High Availability (HA),
- Lastverteilung – Server Load Balancing (SLB).

[118] Als weiterführende Literatur sind die Werke „Cluster Computing“ [5] und „Parallel Programming“ [44] empfehlenswert.
[119] Vgl. [23] S. 169–170 und [5] S. 27.
[120] Vgl. [5] S. 53.
[121] Vgl. [5] S. 54–60.
[122] Vgl. [25]

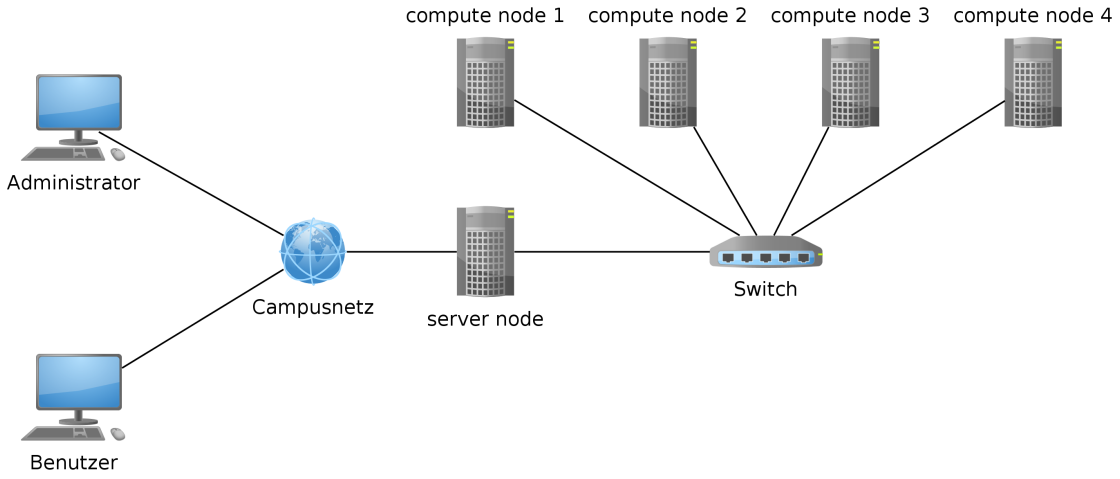

Abbildung 4.1: Grundaufbau eines HPC-Clusters mit 4 Knoten. Vgl. [5] S. 52.

Da die Megware Computer GmbH hauptsächlich HPC-Cluster herstellt, wird weiter auf HPC-Cluster näher eingegangen.

4.2 HPC-Cluster

Die HPC-Cluster sind primär zur Steigerung der Rechenleistung ausgelegt. Sie werden meistens in wissenschaftlichen Bereichen für die Lösung von gut parallelisierbarer und rechenintensiver Probleme eingesetzt.[123]

Die Abbildung 4.2 zeigt die Struktur eines HPC-Cluster. Es stehen 2 eigenständige Server-Knoten zur Verfügung: Management-Knoten und Frontend-Knoten. Ein Management-Knoten erfüllt alle Management-Aufgaben des Clusters. Er ist für Monitoring, Wartung und Verwaltung von Knoten zuständig. Ein Frontend-Knoten ist für Anmeldung und Stapelverarbeitung verantwortlich.[124]

Die Stapelverarbeitungssysteme (engl. *batch systems*) werden für die optimale Aufteilung der Rechenlast gebraucht. Das Stapelverarbeitungssystem nimmt die Rechenaufträge (engl. *batch jobs*) entgegen und verteilt diese an die Rechenknoten weiter. Die Nutzer müssen sich jedoch vorher an dem Frontend-Knoten anmelden, um Rechenaufträge vergeben zu können.[125]

Das Netzwerk besteht aus 3 unabhängigen Netzwerk-Komponenten: [126]

[123] Vgl. [23] S. 170–171.
[124] Vgl. [5] S. 52.
[125] Vgl. [5] S. 151.
[126] Vgl. [26] Projektdokumentation.

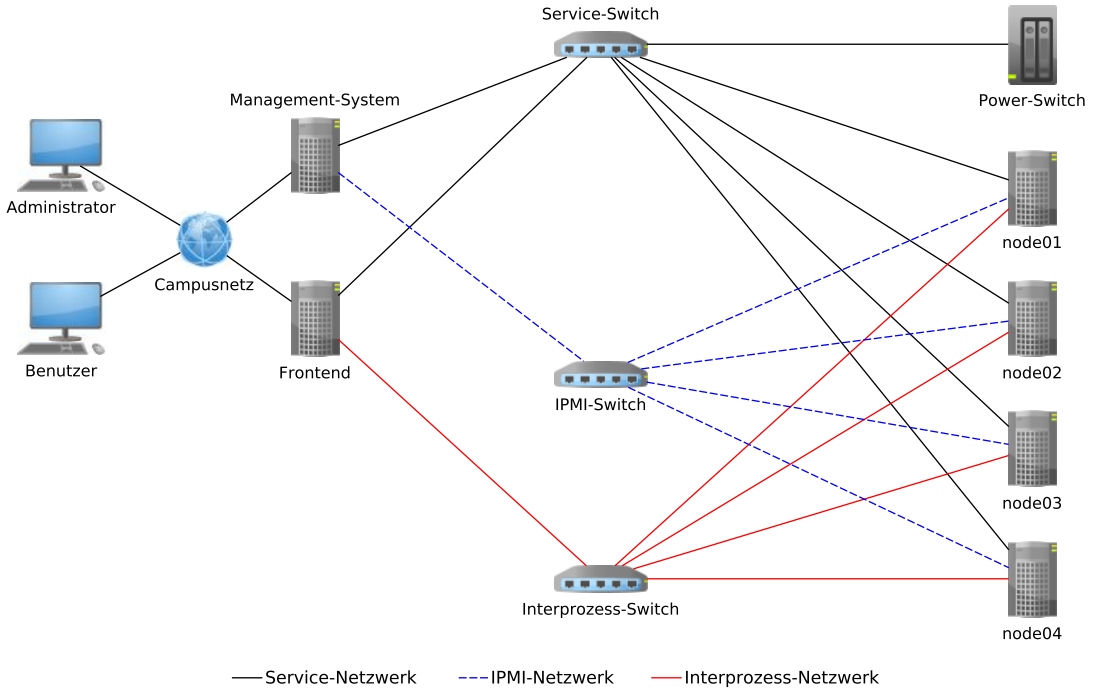

Abbildung 4.2: Beispiel eines HPC-Clusters mit 4 Knoten. Vgl. [26] Projekt-Dokumentation.

- Service-Netzwerk – stellt Dienste zum clusterübergreifenden Management bereit; Managementsysteme benutzen es zur Durchführung von Installationen und Wartung; es kommt meistens 1-Gigabit- bzw. 10-Gigabit-Ethernet zum Einsatz;
- IPMI-Netzwerk – dient dem Management und Monitoring einzelner Knoten; es ermöglicht unter anderem Ein-/Ausschalten, BIOS-Zugriff und Sensoren-Überwachung; für diesen Zweck kann 100-Megabit-Ethernet oftmals ausreichend sein;
- Interprozess-Netzwerk – ermöglicht Kommunikation zwischen Prozessen während der Berechnungen; hier kommt mindestens 10-Gigabit-Ethernet zum Einsatz; bei hohen Anforderungen an die Prozesskommunikation ist eine schnellere und latenzärmere Technologie wie InfiniBand mit Datenübertragungsraten mit bis zu 60 Gbit/s von Vorteil.

Das Stromverteilungssystem, auch PDU (engl. *power distribution unit*) genannt, sorgt für die Bereitstellung der Stromversorgung der Knoten. Es ermöglicht Echtzeitüberwachung von Stromverbrauch, Spannung und Stromstärke. Intelligent managebare Stromschalter (engl. *intelligent switched PDU* bzw. *intelligent power switch*) erlauben außerdem koordiniertes und verzögertes Hoch- und Herunterfahren von Geräten sowie Echtzeitüberwachung von Umgebungsdaten wie Temperatur und Luftfeuchtigkeit.[127]

Abhängig von den Anforderungen können HPC-Cluster durch Grafik-Prozessoren, symmetrische Multiprozessoren, Bandlaufwerke und RAID-Systeme erweitert werden.[128]

[127] Vgl. [26] Projektdokumentation.
[128] Vgl. [26] Projektdokumentation.

4.3 Cluster-Managementsysteme

Bei steigender Knotenzahl ist der Einsatz eines Cluster-Managementsystems naheliegend. Die Installation und Konfiguration eines Clusters werden dadurch stark vereinfacht. Außerdem tragen die Benutzerverwaltung und das Monitoring von wichtigen Knoten-Parametern wie Auslastung, Speicherplatz, Temperatur und Energieverbrauch zum effektiven Cluster-Einsatz bei.

Auf dem Cluster-Markt werden vielfältige Lösungen zum HPC-Cluster-Management angeboten: [129]

- OSCAR (Open Source Cluster Application Resources)[130],
- Rocks Cluster Distribution[131],
- SCore Cluster System Software[132],
- LCFG (Local ConFiGuration system)[133],
- DCC (Debian Cluster Components)[134],
- Bright Cluster Manager[135].

Viele Cluster-Managementsysteme bieten jedoch nur beschränkte Funktionalität, Konfigurierbarkeit und Flexibilität bzw. werden distributions- oder herstellerspezifisch entwickelt. Aus diesem Grund bemüht sich die Megware GmbH um die Entwicklung eigener Cluster-Managementprodukte wie ClustWare-Appliance.

4.4 Megware Cluster-Managementsoftware

Die ClustWare ist modular aufgebaut und basiert auf anderen Soft- und Hardwareprodukten. Die wichtigsten Komponenten sind Appliance-Daemon, Knoten-Daemon, Appliance-Datenbank, Appliance CLI sowie Erweiterungsmodule für Stapelverarbeitung und Knoten-Installation. Siehe Abbildung 4.3.[136]

[129] Vgl. [5] S. 162–164.
[130] Siehe http://svn.oscar.openclustergroup.org/trac/oscar/.
[131] Siehe http://www.rocksclusters.org/.
[132] Siehe http://www.pccluster.org/.
[133] Siehe http://www.lcfg.org/.
[134] Siehe http://dcc.irb.hr/.
[135] Siehe http://www.brightcomputing.com/Bright-Cluster-Manager.php.
[136] Vgl. [27] S. 5–6 und [26] Modulübersicht.

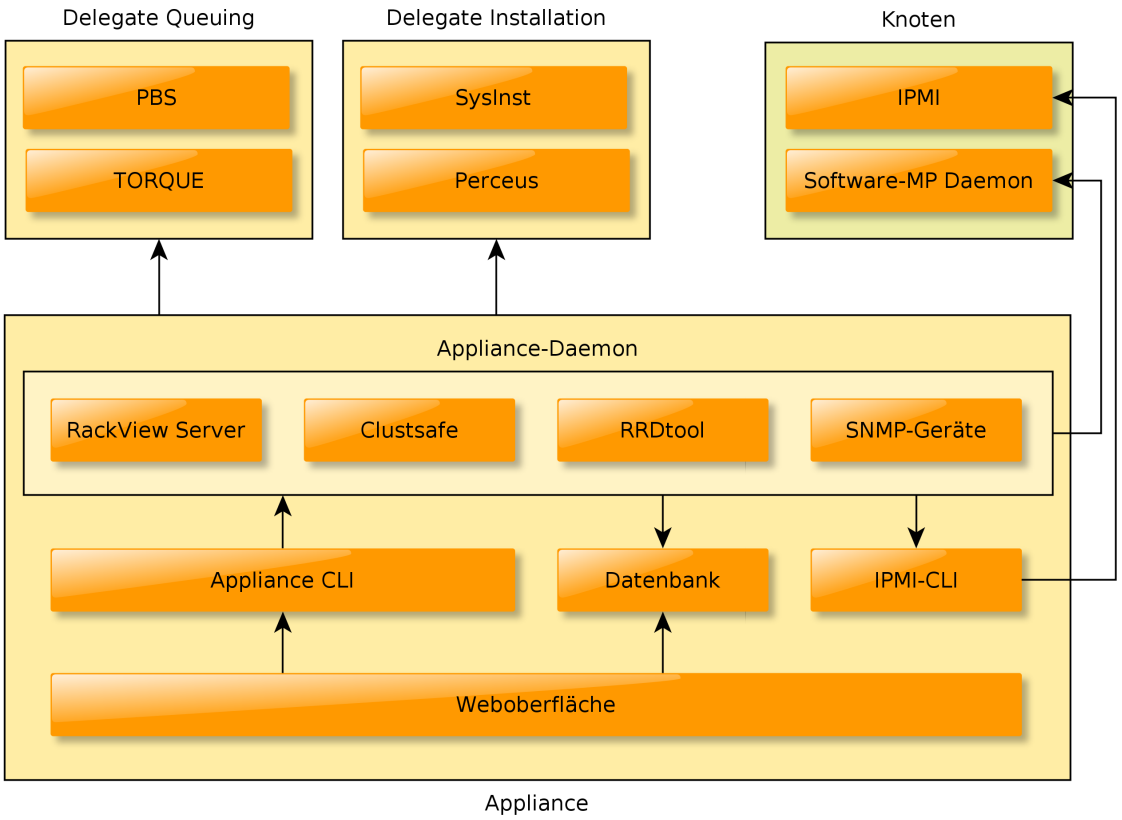

Abbildung 4.3: Appliance Übersicht. Vgl. [26] Modulübersicht.

4.4.1 Appliance-Daemon

Appliance-Daemon ist das Herz der ClustWare und enthält folgende Module zur Administration des Clusters: [137]

- **RackView Server** stellt Daten für das TFT-Bedienpanel RackView zur Verfügung;
- **Clustsafe Modul** verarbeitet PDU-Sensordaten und steuert den Stromverteiler ClustSafe;
- **RRDTool Modul** ist für Monitoring und Archivierung aller Hardwareparameter zuständig;
- **SNMP-Geräte Modul** verwaltet Geräte wie Switches, Power-Switches und Rack Monitoring Systeme, die per SNMP angesteuert werden können.

Auf jedem Knoten läuft ein Daemon, welcher zusammen mit IPMI eigene Hardware- und Softwareparameter verwaltet und nach einer Anfrage seitens des Appliance-Daemons zurückliefert. Die IPMI-Karte sorgt für den Zugriff auf ausgeschaltete Knoten.[138]

[137] Vgl. [26] Modulübersicht.
[138] Vgl. [26] Modulübersicht.

Die Benutzerverwaltung- und Knoten-Einstellungen werden in der Datenbank gespeichert. Der Appliance-Daemon und somit auch die ClustWare werden über das Appliance CLI gesteuert.[139]

Module Delegate Installation und Delegate Queuing stellen Erweiterungs-Interfaces bereit. Damit ist es möglich weitere Softwareprodukte dynamisch anzubinden, unter anderem: [140]

- SysInst – Installationsmodul von Megware,
- Perceus – Installationsmodul, welches Einsatz von Knoten ohne Festplatten (engl. „diskless nodes“) ermöglicht,
- Portable Batch System – ein Batch-System,
- TORQUE – ein weiteres Batch-System.

4.4.2 Appliance CLI

Das Appliance CLI (vom Englischen „command-line interface“ für „Konsole“) bietet eine Schnittstelle zur ClustWare-Steuerung. Es kann mit dem Befehl „appliance“ auf dem Appliance-Knoten gestartet werden. Die Kommunikation mit dem Appliance-Daemon findet über einen lokalen Dateisystem-Socket statt.[141]

Die Befehlssyntax von Appliance CLI sieht wie folgt aus: [142]

```
appliance command::parameter_1::parameter_2::parameter_n
```

Das Kommando und die Parameter werden durch den doppelten Doppelpunkt „::“ getrennt. Die Kommandos sind *help*, *nodes*, *batch* und *tickets*. Sie können durch die Parameter erweitert bzw. ergänzt werden, siehe Listing 4.3.[143]

Die Kommandos *nodes::get* und *nodes::command* unterstützen Wildcards. Somit ist es möglich, gleich für mehrere Geräte die Zustandsinformationen abzufragen bzw. Steuerbefehle auszuführen: [144]

```
# Knoten node01 und node02 ausschalten
appliance nodes::command={node01,node02}::shutdown

# Zeige den Status aller Geraete im Cluster
appliance nodes::get=*::system
```

[139] Vgl. [26] Modulübersicht.
[140] Vgl. [26] Modulübersicht und Projektdokumentation.
[141] Vgl. [27] S. 9.
[142] Vgl. [27] S. 9, 24.
[143] Vgl. [27] S. 24–25.
[144] Vgl. [27] S. 9–10, 26.

```
# Zustandsinformation des Knotens node01 abfragen
appliance nodes::get=node01::infolist

# Ausgabe kann wie folgt aussehen (gekuerzt)
node01::boottime=1317469229
node01::cpu::0::idle=99.88333
node01::cpu::0::user=0.41667
node01::disk::sda1::mountpoint=/boot
node01::disk::sda1::size=103512064
node01::disk::sda1::used=12129280
node01::fan::1=10300
node01::fan::2=8800
node01::memory::buffer=154169344
node01::memory::cached=293863424
node01::memory::free=2217013248
node01::network::eth0::link::state=up
node01::network::eth0::receive::bytes=1355.78333
node01::network::eth0::transmit::bytes=14673.05
node01::pdustate=on
node01::powersupply::12v=11.712
node01::powersupply::3.3v=3.248
node01::powersupply::5v=4.92
node01::system=online
node01::temperature::system=36
node01::uptime=2019407
node01::user::count=0
```

Listing 4.3: Abfragen der Zustandsinformation eines Knotens.

4.4.3 Datenbank

Die Datenbank beinhaltet die Informationen über Gerätezuordnung, Benutzerverwaltung und Monitoring, siehe Abbildung 4.4. Alle Geräte werden in der Tabelle *t_unit* konfiguriert, wobei die Tabelle *t_configclass* die passenden Geräteklassen und die Tabelle *t_rack* das dazugehörige Rack beinhalten. Weiterhin werden in der Tabelle *t_pduport* die Zuordnung von PDU-Ports und in der Tabelle *t_interface* die Netzwerkschnittstellen der jeweiligen Geräte konfiguriert.

In der Tabelle *t_view* wird das Aussehen der Verlaufs-Grafiken zu jeder Geräteklasse konfiguriert.

In der Tabelle *t_config* werden die Grenzwerte festgelegt, die Gerätewerte zugeordnet und benannt. Beim Über- oder Unterschreiten von Grenzwerten werden die Meldungen in der Tabelle *t_alert* erzeugt und ggf. an eine vordefinierte E-Mail-Adresse versandt.

Die E-Mail-Adressen werden in die Tabelle *t_email* eingetragen. Die Nutzerdaten werden in der Tabelle *t_user* organisiert.

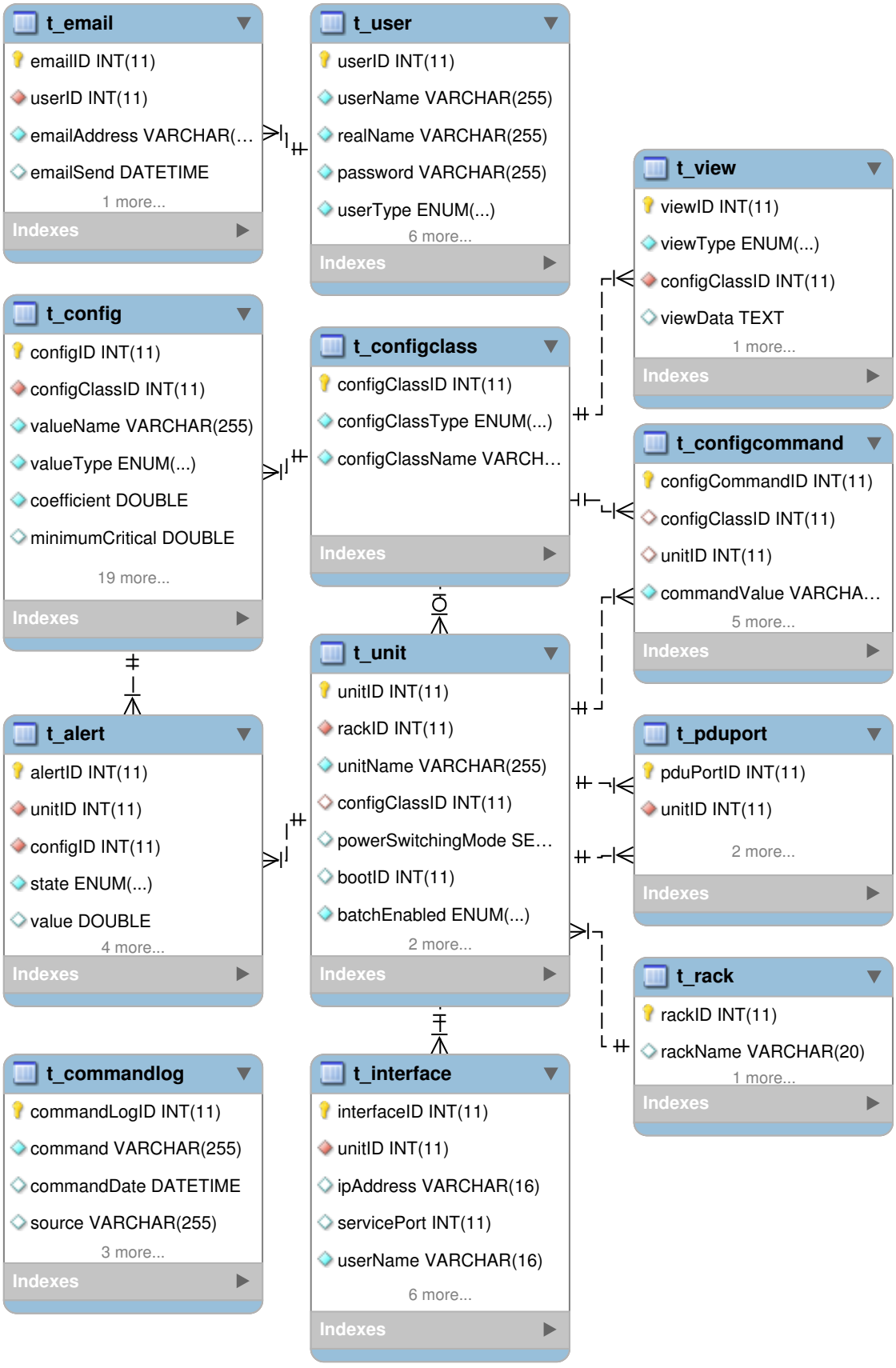

Abbildung 4.4: EER-Diagramm (Martin-Notation) angelehnt an ClustWare-Datenbankstruktur.

Die Kommandos, die z. B. beim Über- oder Unterschreiten von Grenzwerten ausgeführt werden sollen, können gerätespezifisch oder klassenweit in der Tabelle *t_configcommand* definiert werden. Die Tabelle *t_commandlog* führt ein Ereignisprotokoll für alle ausgeführte Befehle.

4.4.4 Benutzeroberflächen

Die nötige Cluster-Übersicht kann bei einer großen Anzahl von Knoten ohne Einsatz einer Benutzeroberfläche nicht gewährleistet werden. Aus diesem Grund wurde für die ClustWare eine spezielle Rails-Weboberfläche entwickelt, die das Cluster-Management effektiver gestaltet.

Der Einsatz von AJAX wurde im Vergleich zu der früheren PHP-Weboberfläche stark reduziert, somit ist eine flüssige Bedienung auch bei über 1000 Knoten gewährleistet. Die Kommunikation zwischen der Weboberfläche und dem Appliance-Daemon verläuft jedoch weiterhin direkt durch den Aplliance CLI Socket. Dies führt z. B. dazu, dass bei jeder neuen Anfrage die Knotendaten auf neu geparst werden müssen.

Ein API, das Zugriffe auf das Appliance CLI und auf die Datenbank kapselt, kann dieses Problem lösen. Dieser Ansatz wurde bei Entwicklung der Qt-Oberfläche verfolgt, das Qt-Framework wurde dabei als Basis für Netzwerk- und Kommunikationsschicht verwendet.

Es hat sich aber gezeigt, dass es nur eine mangelhafte Anbindung von der Programmiersprache Ruby 1.9 an das Qt-Framework 4.7 existiert. Somit lässt sich das in Qt entwickelte API nur für die Qt-Oberfläche und kaum für die Ruby-Weboberfläche verwenden. Daher wird jetzt an einer RESTful-basierten API-Lösung gearbeitet, die in Ruby implementiert werden soll.

5 Entwurf eines RESTful Web Services

Dieses Kapitel befasst sich mit dem Entwurf und der Implementierung des Appliance-API, welches auf der REST-Architektur basiert. Ein API für umfangreiche Cluster-Managementsysteme wie ClustWare soll im Idealfall folgenden Funktionsumfang anbieten:

- Cluster-Installation
- Steuerung und Konfiguration des Clusters,
- Monitoring des Clusters,
- Verwaltung von Batch-Jobs,
- Benutzerverwaltung und Autorisierung.

Im Rahmen dieser Arbeit soll das Cluster-Monitoring vordergründig analysiert werden, da es sich als Schwachpunkt bei früheren Implementierungen erwiesen hat. Weitere Funktionalitäten wie Batch-Verwaltung, Benutzerautorisierung etc. werden dabei nur teilweise angesprochen, sie sollen das API später erweitern.

Das Cluster-Monitoring wird auch bei der Implementierung an erster Stelle untersucht. Somit sollen die entsprechenden Programmschichten Parser, Modell, Controller und View ansatzweise unter der Verwendung des Frameworks Ruby on Rails implementiert werden.

Die Aspekte der Sicherheit und Skalierbarkeit werden anschließend näher betrachtet.

5.1 Vorüberlegungen

Vor dem Erstellen einer neuen Architektur sollen neben der bestehenden Architektur die Lastverteilung analysiert und die Engpässe ermittelt werden. Dadurch wird die mögliche architekturbedingte Performance-Einbuße von vornherein vermieden.

Wie schon früher erwähnt wurde, werden die zahlreichen CLI-Anfragen und das Parsen von den Anfragen als mögliche Ursachen für lange Antwortzeiten vermutet. Um zu bestimmen, wie stark diese Faktoren tatsächlich die Performance beeinflussen, wurden mehrere Laufzeitmessungen durchgeführt, siehe Abbildung 5.1 und Tabelle 5.1.

Die Messung hat einen linearen Zusammenhang zwischen der Laufzeit der Funktionen und der Knotenanzahl gezeigt. Die Anzahl der Durchläufe entspricht dabei der Anzahl der Funktionsaufrufe bei dem Cluster-Monitoring mit der entsprechenden Knotenanzahl.

Funktion \ Durchlauf	1000	2000	3000	4000	5000	6000	7000
CLI (1 Thread)	1,714	4,660	4,744	7,594	9,533	9,653	13,353
CLI (5 Threads)	1,220	2,394	3,558	5,962	6,185	8,260	8,249
Daten Parsen	1,811	3,446	5,160	7,084	8,606	10,792	11,986
XML erzeugen	1,125	2,240	3,340	4,485	5,530	6,662	7,946
JSON erzeugen	0,229	0,449	0,680	0,960	1,127	1,343	1,567
Memcached Set	0,008	0,011	0,016	0,027	0,026	0,038	0,046
Memcached Get	0,029	0,095	0,135	0,178	0,190	0,217	0,248

Tabelle 5.1: Laufzeitmessung der Funktionen (Zeit in Sekunden).

Trotz der Erwartungen sind die CLI-Aufrufe und das Parsen relativ effizient. Durch Parallelisierung der CLI-Aufrufe wurde der Datendurchsatz weiterhin optimiert. Die Funktionen aus der View-Schicht, welche die Repräsentationen erzeugen, sind weniger performant als vorher angenommen. Die Funktionen für das Generieren von XML/HTML beeinflussen das Laufzeitverhalten genauso stark wie CLI-Aufrufe. Die Funktionen, die auf einen Cache (Memcached-Server) zugreifen, sind hingegen sehr performant.

Bei einem Cluster mit 2000 Rechenknoten kann die Antwortzeit wie folgt berechnet werden (Netzwerkschicht wird nicht beachtet):

$$
\begin{aligned}
Antwortzeit_{2000} &= R_{\text{CLI-Abfrage}} + R_{\text{Parsen}} + R_{\text{XML-Generierung}} \\
&= 2,394s + 3,446s + 2,240s \\
&= 8,080s \\
&\approx 8,1s \\
&\qquad\qquad R = Rechenzeit
\end{aligned}
$$

Somit beträgt die Antwortzeit bei einem Cluster mit 2000 Knoten 8,1 Sekunden, bei einem Cluster mit 4000 Knoten 17,5 Sekunden. Bei mehreren gleichzeitigen Anfragen (etwa bei mehreren Clients) wird die Antwortzeit noch weiter ansteigen, was evtl. zur Prozessorüberlastung führen kann.

Um die Antwortzeit und Prozessorauslastung zu minimieren, können die rechenaufwändigen Operationen weiterhin optimiert werden, was allerdings nicht zur dauerhaften Lösung des Problems führt.

Eine weitere Möglichkeit die Systemperformance zu verbessern ist die Abhängigkeit zwischen den Abfragen und den komplexen Berechnungen aufzulösen. Das hat dann eine bessere Gesamt-Performance zur Folge, wobei die Ausführungszeit einzelner Operationen unverändert bleibt.

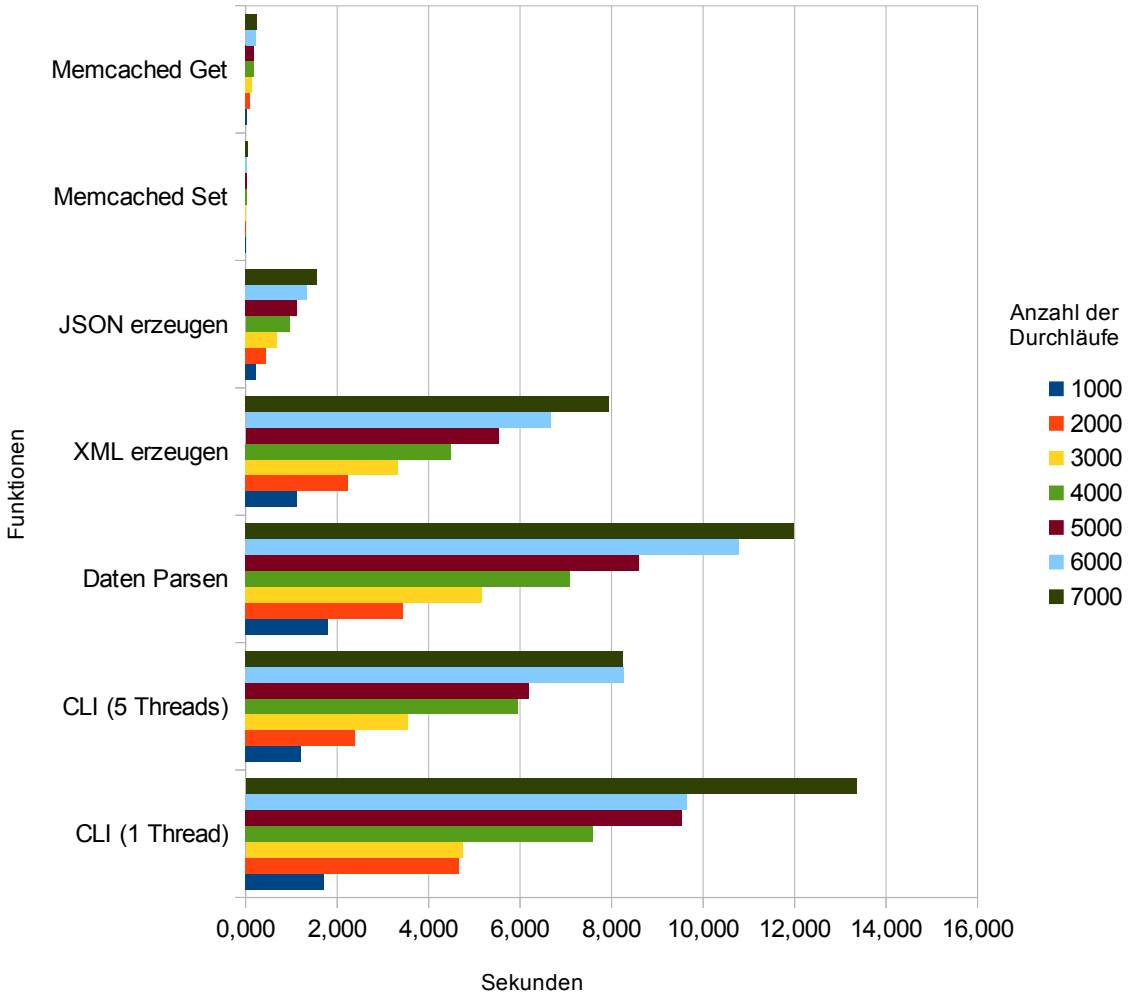

Abbildung 5.1: Laufzeitmessung der Funktionen.

5.2 Architekturentwurf

Das Web Service API soll auf der REST-Architektur basieren. Die Performance hat eine hohe Priorität und soll bei allen Architekturentscheidungen mitberücksichtigt werden. Alle REST-Regeln sollten somit befolgt werden, solange sie sich nicht negativ auf die Performance auswirken.

Das Web Service soll auf einem Webserver unter Einsatz des Ruby on Rails Frameworks laufen. Dadurch wird die Appliance-Architektur durch eine weitere Schicht erweitert, die der Entwurfsmuster-Idee Fassade bzw. Adapter nahe liegt.[145]

Das Web Service API soll mehrere Repräsentationen anbieten, um unterschiedliche Clients mit Daten versorgen zu können. Die Daten werden von der Datenbank und von dem Appliance CLI geholt, geparst und in dem angeforderten Format an Client übergeben.

[145] Vgl. [19] S. 171–172, 212–214.

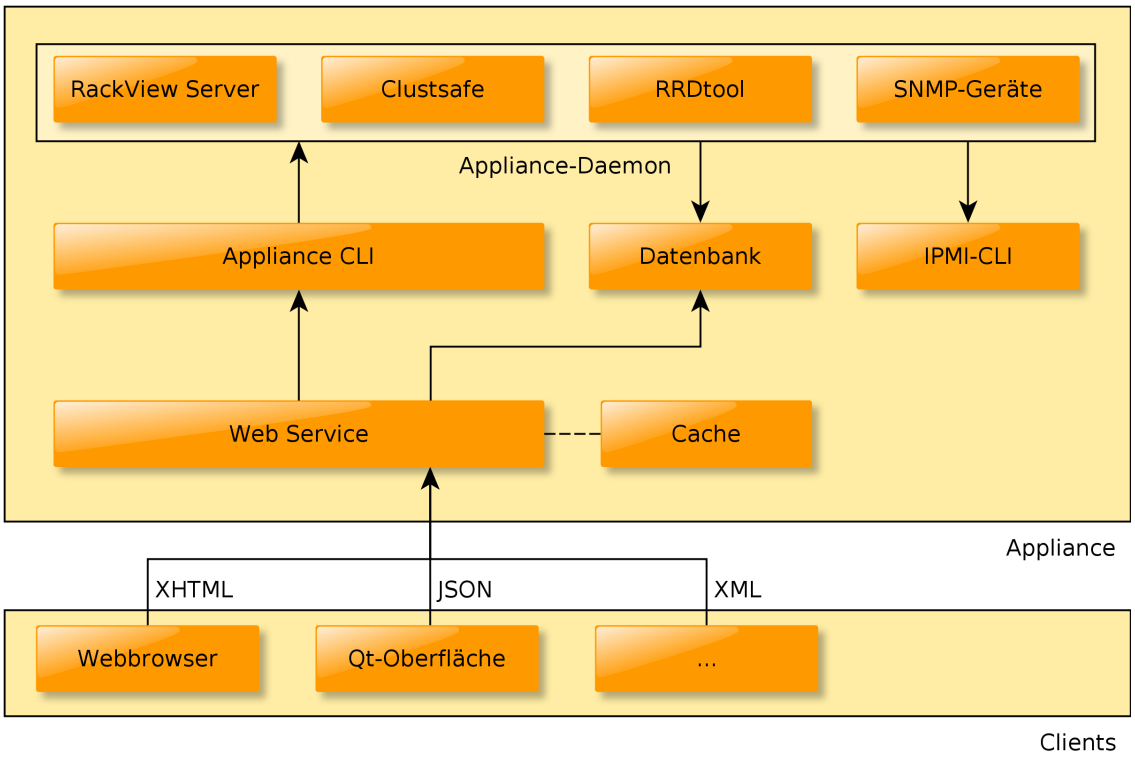

Abbildung 5.2: Erster Architekturentwurf.

Um zeitliche Abhängigkeiten zwischen den Client-Anfragen und der rechenintensiven Operationen aufzuheben, sollen die Ergebnisse der CLI-Aufrufe und des Parsens zwischengespeichert werden. Für diesen Zweck sind die Daten-Caches besonders gut geeignet. Die hohe Zugriffsgeschwindigkeit sorgt für die kurze Antwortzeit der Client-Anfragen, vgl. Diagramm 5.1 (Memcached). Die Schreib- und Leseoperationen sind sehr schnell und effizient, was in einer schnell aktualisierbaren Umgebung sehr wichtig ist. Die Flüchtigkeit des Cache ist dabei unbedeutend, da die Daten temporär sind.

Dank der neuen Architektur, siehe Abbildung 5.2, werden die Daten nur bei der ersten Anfrage berechnet und im internen Daten-Cache gespeichert. Folgt danach eine gleiche oder eine ähnliche Anfrage, werden die Daten aus dem Daten-Cache zurückgegeben, soweit diese noch aktuell sind. Die folgende Rechnung zeigt die Antwortzeit bei der ersten Anfrage (als Daten-Cache wird Memcached verwendet):

$$
\begin{aligned}
Antwortzeit_{2000} &= R_{\text{CLI-Abfrage}} + R_{\text{Parsen}} + R_{\text{XML-Generierung}} + R_{\text{Memchached Set}} \\
&= 2,394s + 3,446s + 2,240s + 0,011s \\
&= 8,091s \\
&\approx 8,1s
\end{aligned}
$$

Die Antwortzeit bei der ersten Anfrage ist unwesentlich angestiegen. Bei weiteren Anfragen entspricht die Antwortzeit der Zeit, die für das Abrufen der serialisierten Monitoring-Daten aus dem Daten-Cache benötigt wird:

$$\begin{aligned} Antwortzeit_{2000} &= R_{\text{Memchached Get}} \\ &= 0,095s \\ &\approx 0,1s \end{aligned}$$

Die Monitoring-Daten stehen somit dem Nutzer in weniger als 0,1 Sekunden zur Verfügung (Netzwerk ist unbeachtet), was 81 mal schneller ist, als zuvor. Auch bei größerer Knotenanzahl steigt dieser Wert unwesentlich, sodass bei 7000 Knoten nur 0,25 Sekunden gebraucht werden, um die Daten aus dem Daten-Cache abzurufen.

Diese Lösung hat leider auch viele Nachteile. Die Antwortzeit der ersten Anfrage lässt sich nicht reduzieren, denn die Daten befinden sich noch nicht in dem Daten-Cache und müssen zuerst berechnet werden.

Weiterhin führen viele Webserver (Apache, Nginx) mehrere Prozesse einer Webanwendung aus, um die Anfragen schnell abzuarbeiten. Somit wird jeder Prozess einen eigenen Cache besitzen. Diese Caches sind redundant und müssen einzeln gepflegt und aktualisiert werden, das ist sehr rechenintensiv.

Um diese Nachteile zu umgehen, soll der Cache aus der Webserver-Umgebung in eine andere Schicht ausgelagert werden.

Die erste Möglichkeit stellt den Einsatz eines Proxy-Server auf der HTTP-Ebene zum Cachen dar.[146] Dabei wird durch einen Expires-Header die maximale Gültigkeit des Cache angegeben, was jedoch einen großen Nachteil nach sich zieht. Die Daten werden nur in bestimmten Zeitabständen aktualisiert. Damit der Cache möglichst optimal arbeitet, müssen diese Zeitabstände von mehreren Minuten bis mehreren Stunden betragen.

Die zweite Möglichkeit ist, den Cache in den Appliance-Daemon auszulagern. Bei dieser Lösung liegt die Schwierigkeit in der Koordination des Schreibzugriffs auf den Cache. Weiterhin bleibt die lange Antwortzeit bei erster Anfrage als Problem bestehen.

Somit ist es zu erkennen, dass durch alleinige Auslagerung des Cache die Performance-Anforderungen nicht zu erfüllen sind. Wie die Abbildung 5.3 zeigt, lassen sich diese Anforderungen mit gleichzeitigem Auslagern von Cache und Parser aus der Webserver-Schicht erreichen.

[146] Vgl. [32] Kapitel 13. Caching in HTTP.

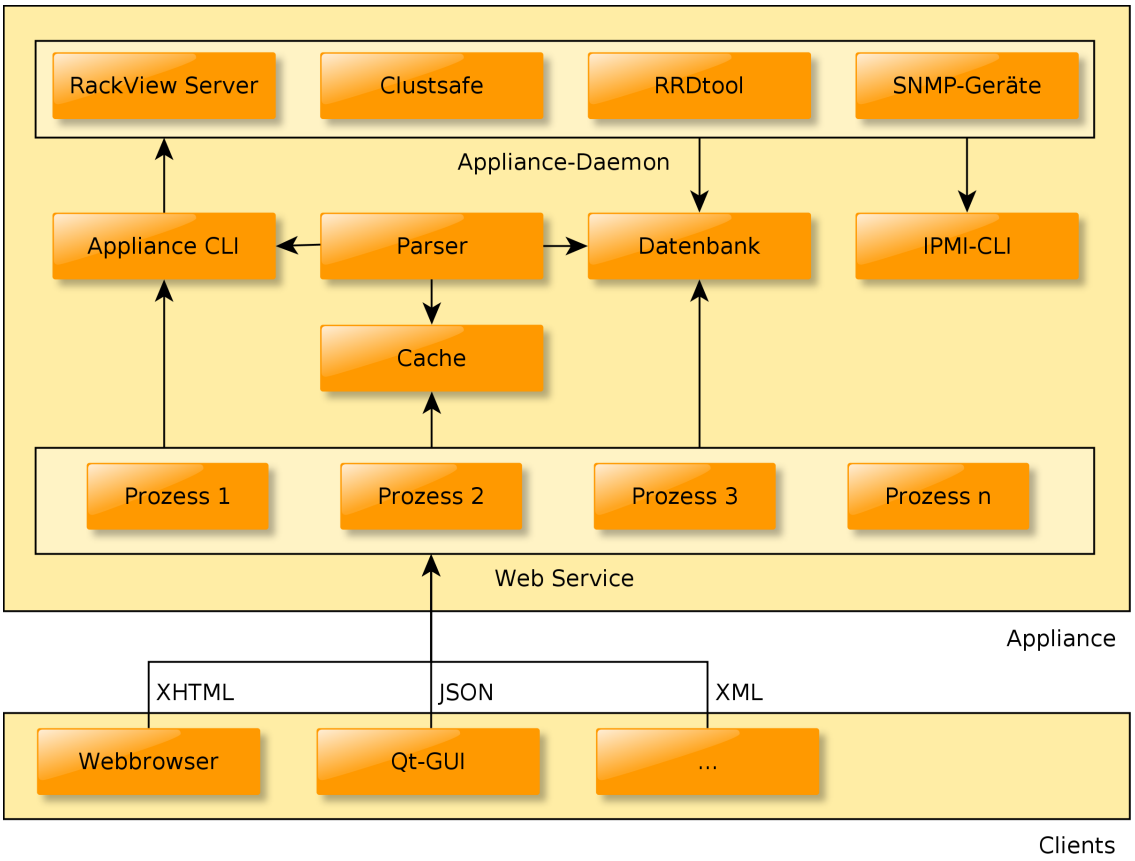

Abbildung 5.3: Zweiter Architekturentwurf.

Der Parser wird teilweise aus dem Web Service ausgelagert, um rechenaufwändige Berechnungen zwischenzuspeichern. Die Daten im Cache werden periodisch durch den Parser aktualisiert. Kommt nun eine Client-Anfrage, die auf die Ergebnisse der rechenaufwändigen Berechnungen zugreifen will, werden die Daten nur aus dem Cache abgeholt und direkt an den Client weitergereicht.

Anderenfalls werden die Anfragen von dem Web Service durch den Zugriff auf Appliance CLI und Appliance-Datenbank beantwortet, falls das Zwischenspeichern der Daten nicht möglich ist bzw. sich nicht lohnt.

Hiermit werden alle Client-Anfragen innerhalb von 0,1 Sekunde (beim Cluster mit 2000 Knoten) beantwortet. Die Aktualität der Daten ist dabei von der Periodizität der Aktualisierung des Parsers abhängig. Auf die Aktualisierung der Daten wird im Kapitel 5.3 näher eingegangen.

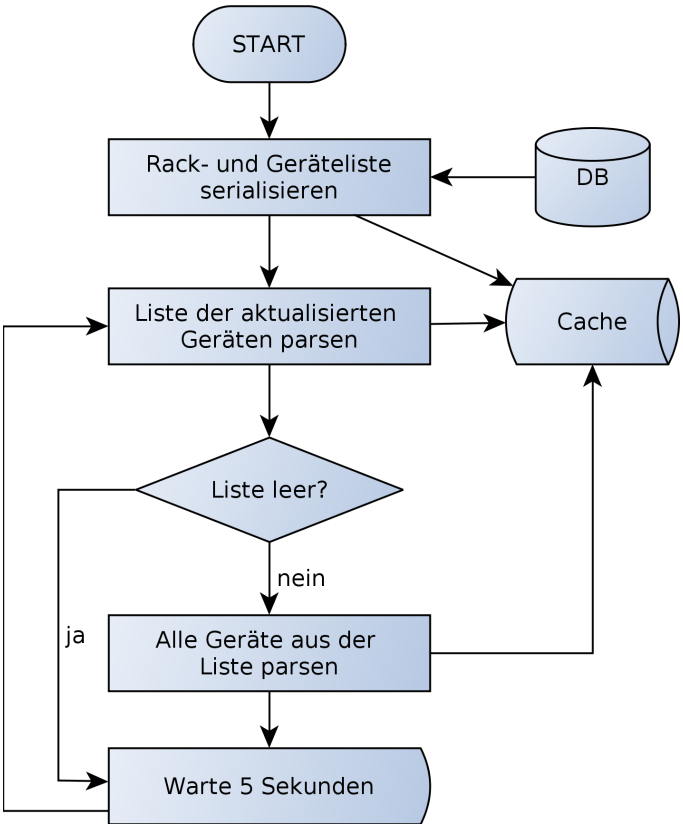

Abbildung 5.4: Flussdiagramm des Parsers.

5.3 Parser

Der Parser wird in der Appliance als ein weiterer Daemon definiert. Die Hauptaufgabe des Parsers ist, die Monitoring-Daten des Clusters möglichst aktuell im Daten-Cache zu halten.

Die Konfiguration des Clusters wird bei der Initialisierung aus der Datenbank gelesen, serialisiert und im Cache gespeichert, siehe Abbildung 5.4. Darauf folgt die Überprüfung, welche Gerätedaten im Cache veraltet sind. Diese Gerätedaten werden dann wie folgt aktualisiert:

1. eine CLI-Abfrage wird erstellt und ausgeführt;
2. das Ergebnis der Abfrage wird in ein Hash-Objekt geparst;
3. das Hash-Objekt wird serialisiert und im Cache gespeichert.

Der Parser wird in der Programmiersprache Ruby implementiert. Sollte sich später herausstellen, dass der Parser die Monitoring-Daten nicht effizient genug verarbeitet, so kann das Parsen in der Programmiersprache C implementiert und in den Appliance-Daemon als ein Modul integriert werden.

CLI-Abfragen ausführen

Das Appliance CLI ermöglicht die Kommunikation durch einen UNIX-Domain-Socket. Somit können die Anfragen direkt (ohne Terminal) von Appliance CLI beantwortet werden.

Ruby stellt eine Bibliothek für den Zugang zu den Netzwerkdiensten auch auf einer niedrigen Ebene zur Verfügung. So unterstützt die UNIXSocket-Klasse die Interprozesskommunikation mit Hilfe des streambasierten Unix-Domain-Protokolls, siehe Listing 5.1.[147]

```
1  require 'socket'
2
3  SOCKET = "/tmp/applianceSocket" # Socket-Pfad
4  device = "node01"               # Geraet
5  result = ""                     # Ergebnis der Abfrage
6
7  # Verbindung zum Appliance-Socket aufbauen
8  client=UNIXSocket.open(SOCKET)
9
10 # Zustandsinformation zum Geraet abfragen
11 client.send("nodes::get=#{device}::infolist\n\n\n", 0)
12
13 # Ergebnis der Abfrage abholen
14 while (tmp=client.recvfrom(1024)[0]).length>0
15   result << tmp
16 end
17
18 # Verbindung zum Socket schliessen
19 client.close
```

Listing 5.1: CLI-Abfrage

Hash erstellen und serialisieren

Nachdem eine Abfrage erfolgreich abgearbeitet wurde, wird das Ergebnis in ein temporäres Hash umgewandelt. Das temporäres Hash wird analysiert und es wird daraus ein weiteres Hash, Gerät-Hash, generiert. Das Gerät-Hash kann dann anschließend serialisiert und in einem Cache gespeichert werden, siehe Listing 5.2.

Obwohl das doppelte Erstellen eines Hashes ineffizient ist, lässt es sich an viele Daten-Variationen anpassen. Der Parser ist somit an künftige Änderungen der Namenskonvention bei den Abfragen gut vorbereitet. Bei einer schlechten Performance kann dieses Verfahren optimiert werden, sodass nur eine begrenzte Anzahl der Daten-Variationen unterstützt wird.

[147] Vgl. [51] S. 811, 891–892.

Für die Serialisierung der Daten wird das Format JSON eingesetzt. Das Serialisieren und Deserialisieren in JSON nimmt wesentlich weniger Zeit in Anspruch, als in den Formaten XML oder YAML. Obwohl die Marshal-Bibliothek noch bessere Performance aufweisen kann, leidet dadurch die Plattformunabhängigkeit, was den Austausch von Daten beeinträchtigen kann.[148]

Die Bibliothek YAJL wird anstatt der Standard-JSON-Bibliothek aufgrund einer besseren Performance verwendet.[149]

```
1  require 'yajl'
2
3  t_hash = {}              # temporaeres Hash
4  device_hash = {}         # Geraet-Hash
5
6  # temporaeres Hash erstellen
7  result.each_line do |i|
8    temp = i.split("=")
9    t_hash[temp.first] = temp.last
10 end
11
12 # Geraet-Hash erstellen
13 if t_hash.has_key?("#{id}::hostname")
14   device_hash[:hostname] = t_hash["#{id}::hostname"]
15 end
16 if t_hash.has_key?("#{id}::boottime")
17   device_hash[:boottime] = t_hash["#{id}::boottime"]
18 end
19 ...
20 # Geraet-Hash serialisieren
21 device = Yajl::Encoder.encode(device_hash)
```

Listing 5.2: Hash erstellen und serialisieren

Cache aktualisieren

Als Implementierung für den Cache-Server kommt Memcached zum Einsatz. Memcached ist ein hochperformantes, verteiltes Objekt-Caching-System. Es wird oftmals zur Erhöhung der Leistung von dynamischen Web-Anwendungen eingesetzt.[150]

Die Kommunikation mit dem Memcached-Server kann entweder über Unix-Domain-Socket oder über die Protokolle TCP/UDP stattfinden. Die abgespeicherten Daten werden mit einem eindeutigen Schlüssel versehen und als Zeichenketten im Arbeitsspeicher abgelegt, siehe Listing 5.3.[151]

[148] Vgl. [30] und [48].
[149] Vgl. [24].
[150] Vgl. [28].
[151] Vgl. [29] Configuring Memcached.

Komplexere Datenstrukturen wie Objekte oder Hashs müssen im Vorfeld serialisiert werden. Daten können dauerhaft oder zeitweise abgespeichert werden, siehe Zeile 17 in Listing 5.3.[152]

Bei der Aktualisierung des Cache wird für jedes Gerät ein Entity-Tag (Version der Aktualisierung) und ein Zeitstempel der letzten Änderung gespeichert. Damit wird eine redundante Datenübertragung vermieden. Auf die Verwendung des ETags und des Zeitstempels wird im Kapitel 5.4.2 näher eingegangen.

```
1  require 'memcached'
2  require 'yajl'
3
4  # Verbindung zum Cache-Server aufbauen
5  CACHE = Memcached.new("localhost:11211", :no_block => true,
6                        :buffer_requests => false,
7                        :noreply => true,
8                        :binary_protocol => false)
9
10 # Geraet-Hash serialisieren
11 device_json = Yajl::Encoder.encode(device_hash)
12
13 # Geraet im Cache aktualisieren
14 CACHE.set "device:#{device_hash['id']}:json", device_json
15
16 # Zeitstempel der letzter Aenderung setzen
17 modified = Time.now.getutc
18 CACHE.set "device:#{device_hash['id']}:modified", modified
19
20 # ETag fuer 4 Minuten setzen
21 CACHE.set "device:#{device_hash['id']}:etag", etag, 240
```

Listing 5.3: Cache aktualisieren

5.4 Ressourcen und Routen

Die Funktionalität des Web Service API wird auf die Ressourcen abgebildet, siehe Tabelle 5.2. Die Ressourcen werden in 2 Bereiche unterteilt. Die Primärressourcen aus dem Monitoring-Bereich sind Racks, Devices und Jobs. Die Primärressourcen aus dem Konfigurations-Bereich sind Racks, Devices und PDUs.

Alle Primärressourcen stellen entsprechende Listenressourcen, die nur eine Attributuntermenge der Primärressourcen beinhalten, bereit. Die Monitoring-Primärressource Devices besitzt die Subressource Charts. Die Aktivitätsressourcen Tickets und Meldungen stellen die prozessbedingten Ressourcen dar, welche auch als Subressourcen bei Monitoring-Primärressourcen Devices und Racks betrachtet werden können. Bei allen Ressourcen wird vorerst auf das Filtern und Paginierung verzichtet.

[152] Vgl. [29] What Memcached Is.

Ressource / Methode	GET	PUT	DELETE
/	Listet Links zu den Ressourcen auf	-	-
/racks	Listet alle Racks auf	-	-
/racks/{id}	Zeigt den Rack mit der ID=id an	-	-
/devices	Listet alle Devices auf	-	-
/devices/{id}	Zeigt den Device mit der ID=id an	-	-
/devices/{id}/charts/{cid}	Zeigt das Device-Diagramm mit der DeviceID=id und der ChartID=cid an	-	-
/jobs	Listet alle Batch-Jobs auf	-	-
/jobs/{id}	Zeigt den Batch-Job mit der ID=id an	Ändert den Batch-Job mit der ID=id	Löscht den Batch-Job mit der ID=id
/tickets	Listet alle Tickets auf	-	-
/tickets/{id}	Zeigt den Ticket mit der ID=id an	-	-
alerts/	Listet alle Meldungen auf	-	-
/config	Listet Links zu den Konfigurations-ressourcen auf	-	-
/config/devices	Listet alle Devices auf	-	-
/config/devices/{id}	Zeigt den Device mit der ID=id an	Ändert den Device mit der ID=id oder fügt einen neuen Device mit der ID=id hinzu	Löscht den Device mit der ID=id
/config/racks	Listet alle Racks auf	-	-
/config/racks/{id}	Zeigt den Rack mit der ID=id an	Ändert den Rack mit der ID=id oder fügt einen neuen Rack mit der ID=id hinzu	Löscht den Rack mit der ID=id
/config/pdus	Listet alle PDUs auf	-	-
/config/pdus/{id}	Zeigt die PDU mit der ID=id an	Ändert die PDU mit der ID=id oder fügt eine neue PDU mit der ID=id hinzu	Löscht die PDU mit der ID=id
/config/users	Listet alle Benutzer auf	-	-
/config/users/{id}	Zeigt den Benutzer mit der ID=id an	Ändert den Benutzer mit der ID=id	Löscht den Benutzer mit der ID=id

Tabelle 5.2: Ressourcen und ihre Routen.

5.4.1 Models

Nachfolgend werden nur die Primär- und Listenressourcen Racks und Devices aus dem Monitoring-Bereich betrachtet. Alle anderen Ressourcen können analog implementiert werden und sind für die Beurteilung der Qualitäten wie Performance oder Datendurchsatz irrelevant.

MRack

Das Model MRack vertritt die Primär- und Listenressource Racks. Da Ruby vollständig objektorientiert ist, stellen alle Datentypen Objekte dar. Somit sind die Klassen auch Objekte mit ihren eigenen Attributen und Methoden. Diese Eigenschaft wird wie folgt bei der Implementierung eingesetzt:

- die Attribute (Präfix @@) und Methoden (Präfix self) der MRack-Klasse beziehen sich auf die Listenressource Racks, siehe Zeilen 3–12 in Listing 5.4;
- die Attribute und Methoden des MRack-Objektes beziehen sich auf die Primärressource Racks, siehe Zeilen 14–22 in Listing 5.4.

```
1  class MRack
2
3    #Holt alle Racks aus dem Memcached ab.
4    def self.all
5      @@json = CACHE.get("racks")
6      self
7    end
8
9    #Gibt die Rack-Liste im JSON-Format zurueck.
10   def self.to_json attribute
11     @@json
12   end
13
14   #Holt das Rack mit der ID=id aus dem Memcached ab.
15   def initialize(id)
16     @json = CACHE.get("racks:#{id}")
17   end
18
19   #Gibt das Rack im JSON-Format zurueck.
20   def to_json attribute
21     @json
22   end
23
24 end
```

Listing 5.4: Model MRack (gekürzt)

MDevice

Ähnlich wie das Model MRack vertritt das Model MDevice die Primär- und Listenressource Devices. Die Device-Listenressource wird durch die Klassen-Methoden *self.all* und *self.where* berechnet. Dabei werden nach Möglichkeit nur Teilaktualisierungen bzw. keine Aktualisierungen durchgeführt. Das soll die Performance des Web Services zusätzlich verbessern und wird im nächsten Kapitel näher betrachtet.

Bei der Methode *self.all* wird zuerst die Liste aller Geräte ermittelt, siehe Zeile 5 in Listing 5.5. Anschließend werden die aktuellen Monitoring-Daten für jedes Gerät vom Memcached-Server abgeholt.

```
1   #Holt alle Devices vom Memcached-Server ab.
2   def self.all
3
4     #Holt die Device-Liste ab.
5     device_list = Yajl::Parser.parse CACHE.get("devices")
6
7     #Holt die Monitoring-Daten fuer jeden Device
8     #aus der Device-Liste ab.
9     devices = []
10    device_list.each{|id|  #Vorbereite Memcached-Multiget.
11      devices << "device:#{id}:json:small"
12    }
13    @@json_array = []
14    CACHE.get(devices).values.each{|dev|
15      @@json_array << dev  #Memcached-Multiget ausfuehren.
16    }
17    self
18  end
```

Listing 5.5: Erster Auszug aus dem Model MDevice

Die Methode *self.where* berechnet zuerst die Liste aller Geräte, die seit letzter clientseitiger Aktualisierung geändert wurden. Die Berechnung der letzten Aktualisierungen findet im Restklassenring $\mathbb{Z}_{RING}$ statt.

Die sinnvolle Tiefe der verfolgten Aktualisierungen wird durch die Konstante *UPDATES_DEEP* festgelegt. Bei Überschreitung dieser Tiefe werden alle Geräte aus dem Cache abgeholt. Anschließend werden die aktuellen Monitoring-Daten für jedes Gerät aus der Liste von dem Memcached-Server abgeholt.

```
1   #Holt die Device-Liste abhaengig von dem ETag-Attribut.
2   def self.where attributes={}
3
4     #Berechnet die Device-Liste.
5     measure_list = []
6     m_etag = @@etag < attributes[:etag] ? @@etag+RING : @@etag
7     if ((m_etag - attributes[:etag]) % RING < UPDATES_DEEP)
```

```
8         (attributes[:etag]+1..m_etag).to_a.each { |t|
              measure_list << "measure:#{t%RING}" }
9       else
10        return MDevice.all
11      end
12
13      #Holt die Monitoring-Daten fuer jedes Device
14      #aus der Device-Liste ab, siehe self.all.
15      ...
16    end
```

Listing 5.6: Zweiter Auszug aus dem Model MDevice

5.4.2 Controller

Nachfolgend wird der Controller MDevicesController betrachtet. Andere Controller wie MRackController weisen eine ähnliche Funktionsweise auf und werden in dieser Arbeit nicht besprochen.

MDevicesController übernimmt die Rolle eines Vermittlers zwischen dem Model MDevice und der Views, welche verschiedene Repräsentationen der Ressource Device anbieten. Obwohl die Zugriffe auf diese Schichten optimiert sind und die Anfragen teilweise zwischengespeichert werden, sollen die redundante Berechnung und damit verbundene Datenübertragung trotzdem verhindert werden. Dies wird durch die HTTP-Header ETag und Last-Modified erreicht.

Der ETag-Header enthält den Wert des Entity-Tags, welches durch die Inode-Methode unter Beachtung des Zeitpunktes der letzten Dateiänderung berechnet wird. Weiterhin ist es möglich, den Datei-Hashwert bzw. die Datei-Versionsnummer zur Berechnung des ETag miteinzubeziehen. Da die Performance des Web Services von großer Bedeutung ist, wird nur die Versionsnummer der letzten Device-Aktualisierung bei der ETag-Berechnung beachtet.[153]

Weiterhin wird die Ermittlung der Aktualität der Daten durch Last-Modified-Header unterstützt. Bei einem Last-Modified-Header wird der Zeitpunkt der letzten Datenänderung übertragen. Obwohl die Header ETag und Last-Modified die gleiche Funktion erfüllen, empfiehlt RFC 2616 beides zu senden.[154]

Um auch zum Teil redundante Übertragung der Monitoring-Daten bei Listenressource Devices zu verhindern, wird ein Aktualitätsfilter eingesetzt. Dabei übermittelt der Client die Versionsnummer der zuletzt erhaltener Daten an den Server. Der Server übermittelt dann nur die Daten, die seither geändert wurden, siehe Abbildung 5.5.

[153] Vgl. [32] Kapitel 14.19 ETag und Kapitel 14.26 If-None-Match.

[154] Vgl. [32] Kapitel 13.3.1 Last-Modified Dates, Kapitel 14.25 If-Modified-Since und Kapitel 13.3.4 Rules for When to Use Entity Tags and Last-Modified Dates.

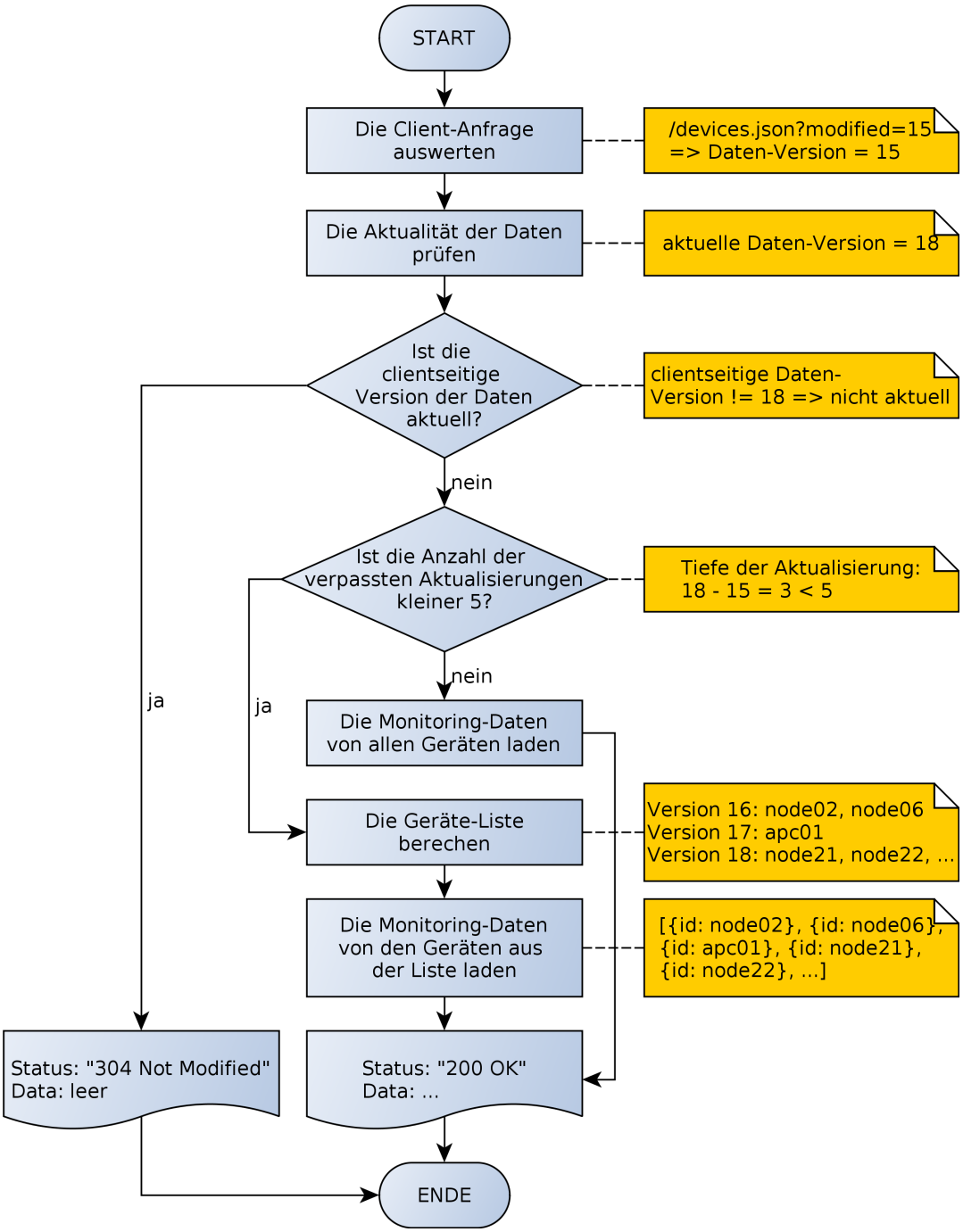

Abbildung 5.5: Auswertung der Anfragen für Listenressource Devices.

Unabhängig davon, welche Methoden der Client zur Vermeidung der redundanten Datenübertragung verwendet, wird die Anfrage mit dem HTTP-Status „304 Not Modified“ beantwortet, falls die clientseitigen Daten noch aktuell sind. In diesem Fall findet keine Übertragung der Monitoring-Daten statt, was die Performance enorm verbessert. Wird dagegen festgestellt, dass die clientseitigen Daten nicht mehr aktuell sind, wird die Anfrage mit dem HTTP-Status „200 OK“ beantwortet und die veränderte Daten an den Client gesendet.

5.5 Repräsentationen

Der Client kann dank dem REST-Architekturstil zwischen verschiedenen Repräsentationen auswählen. So werden Repräsentationen im JSON-, XML- und XHTML-Format angeboten. Das gewünschte Repräsentationsformat wird dabei durch den HTTP-Accept-Header festgelegt. Weiterhin können menschliche Benutzer das Format in der URI festlegen.

5.5.1 JSON

JSON ist ein kompaktes textbasiertes Datenaustauschformat, welches auf einer Untermenge der Programmiersprache JavaScript basiert. JSON bietet als ein Datenaustauschformat zwischen Anwendungen viele Vorteile: [155]

- verfügbar in beinahe allen verbreiteten Programmiersprachen (C, C++, C#, Objective C, Java, Ruby, Python, Perl, PHP, ...),
- gute bis sehr gute Performance,
- einfacher Syntax,
- geringer Overhead.

Aus diesem Grund wird JSON beim Zwischenspeichern von Monitoring-Daten im Cache verwendet und sollte nach Möglichkeit auch bei den Clients bevorzugt werden. Weiterhin wird auf das erneute Parsen und Encodieren von Monitoring-Daten in der View-Schicht verzichtet, was für zusätzliche Performance-Verbesserungen bei der Übermittlung von JSON-Repräsentationen sorgt.

Um Verbesserungen der Performance zu erreichen, wurde das MVC-Architekturmuster von Rails nicht strikt umgesetzt, sodass der JSON-View teilweise in den Parser ausgelagert wurde.

5.5.2 XML

Als ein weiteres Repräsentationsformat wird XML angeboten. XML ist eine Auszeichnungssprache, welche die Daten in einer hierarchisch strukturierten Textform darstellt. Das XML-Format ist weit verbreitet und ist genauso plattformunabhängig und gut unterstützt wie JSON.[156] Es besitzt jedoch einen relativ hohen Overhead und wird in Ruby nicht so effizient wie JSON verarbeitet.

[155] Vgl. [20].
[156] Vgl. [56].

Zum Generieren von XML-Repräsentationen wird die XML-Builder-Bibliothek eingesetzt. Mit einem XML-Builder lassen sich die XML-Dateien mit einem einfachen Ruby-Code generieren, siehe Listing 5.7.[157]

```
1  xml.instruct!  #<?xml version="1.0" encoding="UTF-8"?>
2  xml.racks {|d| #<racks>
3    @racks.racks.each{|rack,devices|
4      d.rack(id: rack) {    #<rack id="Rack1">
5        devices.each {|dev|
6          d.device(id: dev)   #<device id="node01" />
7        }
8      }                     #</rack>
9    }
10 }              #</racks>
```

Listing 5.7: XML-Builder für Listenressource Racks

5.5.3 XHTML

Zur Generierung von XHTML-Repräsentationen bietet Rails das Template-System ERB an. Die ERB-Templates sind sehr flexibel einsetzbar und können beliebige Textdateien (JSON, XML, XHTML, CSV) generieren. Dabei wird der Ruby-Code ähnlich wie bei JSP oder PHP eingebettet, siehe Listing 5.8.[158]

```
1 <div class="rack">
2   <h2><%= @rack.id %></h2>
3   <% @rack.devices.each do |device| %>
4     <%= link_to device, :controller => "m_devices",
5         action: "show", id: device %><br>
6   <% end %>
7 </div>
```

Listing 5.8: ERB-Template für die Ressource Racks

Die ERB-Templates sind jedoch weniger performant, als JSON-Serialisierung bzw. XML-Builder. Aus diesem Grund wird bei Generierung von XHTML-Repräsentationen auf die Einbettung der Monitoring-Daten verzichtet. Stattdessen werden die Monitoring-Daten im JSON-Format per AJAX-Anfrage nachgeladen und per JavaScript in XHTML transformiert.

Bei der Aktualisierung der Monitoring-Daten wird die AJAX-Anfrage erneut gestartet und es wird geprüft, ob die Daten inzwischen geändert wurden (HTTP-Status „200 OK“). Ist das der Fall, wird die XHTML-Repräsentation clientseitig aktualisiert, siehe Abbildung 5.6.

[157] Vgl. [31] Kapitel 8.7 Alternative Template-Systeme.
[158] Vgl. [31] Kapitel 8.1 ERB-Templates.

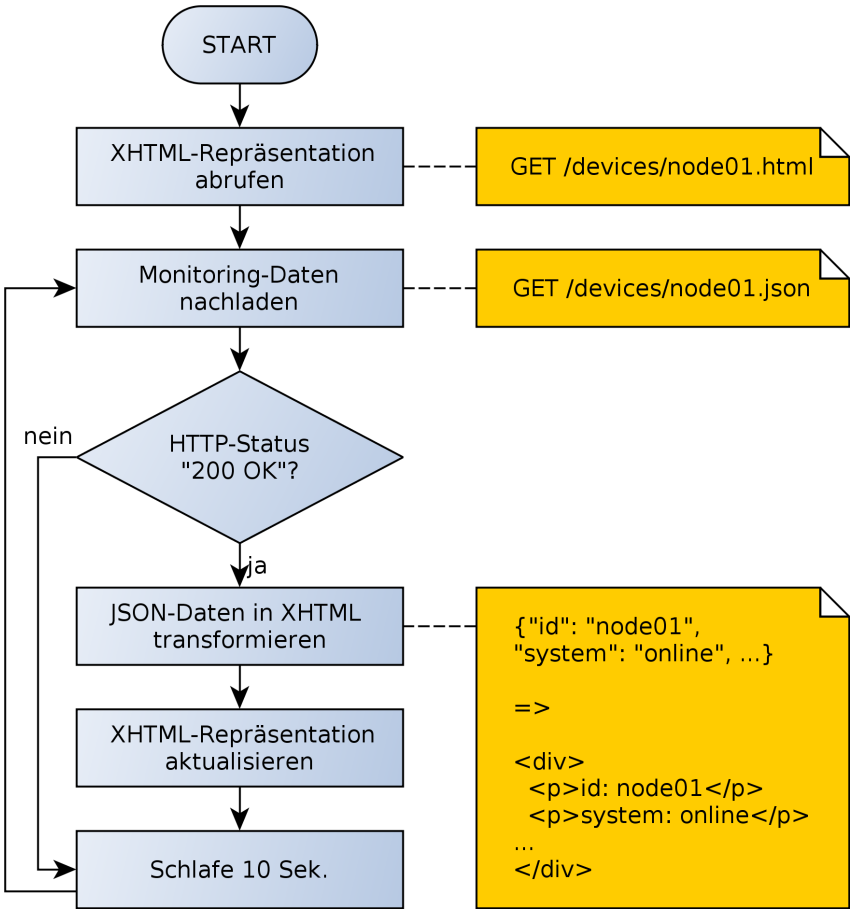

Abbildung 5.6: Aktualisierung der Monitoring-Daten durch AJAX.

5.6 Sicherheit

Da die HPC-Cluster vorwiegend für die Forschung in großen Institutionen eingesetzt werden, spielt die Sicherheit eine wichtige Rolle. Zurzeit werden die meisten Cluster-Systeme in sicheren und geschützten Netzwerkumgebungen verwaltet, wo externe Zugriffe nur sehr begrenzt erlaubt sind.

Die externen Zugriffe werden jedoch bei der Einführung von Web Service API sicherlich noch an Bedeutung gewinnen. Somit kann die Steuerung eines Clusters auch von portablen Geräten wie Smartphones oder Tablets stattfinden.

Damit das geforderte Sicherheitsmaß erreicht werden kann, müssen eine sichere Authentifizierung genauso wie eine verschlüsselte Datenübertragung bei dem Web Service vorausgesetzt werden.

5.6.1 Authentifizierung und Autorisierung

Die Benutzer dürfen einen Cluster nur dann bedienen, falls sie sich vorher authentifiziert haben. Dabei sollen die Zugangsdaten des Benutzers wie Benutzername und Passwort mit den Daten in der Datenbank verglichen werden. Falls die Identität des Benutzers bestätigt wurde, wird der Benutzer authentifiziert, andernfalls nicht.

Weiterhin sollen die Berechtigungen des Benutzers bei allen Interaktivitäten überprüft werden, denn nur die autorisierten Benutzer dürfen die entsprechenden Aktivitäten ausführen. Die Rechte der Benutzergruppe, die in der Datenbank definiert ist, sollen dabei der angeforderten Aktivität entsprechen. Die Gruppen-Rechte werden statisch in den jeweiligen Anwendungen definiert, wobei es vorteilhaft sein kann, die Berechtigungen in der Datenbank festzulegen.

Rails bietet viele Authentifizierungslösungen an, die man in 2 Kategorien unterteilen kann: sessionbasierte und HTTP-basierte Authentifizierung. Die sessionbasierte Authentifizierung ist vor allem für menschliche Benutzer, die auf den Web Service direkt mit dem Webbrowser zugreifen, gut geeignet.[159]

Für computergesteuerte Clients sind die HTTP-basierten Authentifizierungsmethoden wie Basic Authentication oder Digest Access Authentication besser geeignet. Sie werden jedoch durch die Webbrowser nicht optimal unterstützt. So gibt es z. B. keine Möglichkeit sich abzumelden bzw. das Login-Formular lässt sich nur über Umwege umgestalten. Trotz dieser Schwierigkeiten wird die Basic Authentication wegen ihrer Einfachheit und großer Verbreitung zur Authentifizierung eingesetzt, siehe Listing 5.10.[160]

```
1  #Basisklasse fuer alle Controller.
2  class ApplicationController < ActionController::Base
3    #Pruefe vor jeder Aktion die Authentifizierung.
4    before_filter :authenticate
5
6    #Ueberpruefe die Authentifizierung des Benutzers oder
7    #authentifiziere den Benutzer.
8    def authenticate
9      authenticate_or_request_with_http_basic('ClustWare-
           Administration') do |username, password|
10     #Den Benutzer finden und den Passwort ueberpruefen.
11     @user = User.find_by_name!(username)
12     @user.has_password?(password)
13     end
14   end
15 end
```

Listing 5.9: Basic Authentication bei ApplicationController.

[159] Vgl. [42] Action Controller Overview: 10 HTTP Authentications und [39].
[160] Vgl. [40] und [4].

Weiterhin bietet Ruby on Rails ganze Frameworks wie Authlogic oder Device an, welche die Aufgaben der Authentifizierung und der Autorisierung übernehmen, an. Diese Frameworks ziehen jedoch weitreichende Veränderungen der bestehenden Sicherheitsstruktur nach sich, sodass alle ClustWare-Anwendungen angepasst werden müssten.[161]

5.6.2 Verschlüsselte Datenübertragung

Ein großer Nachteil der Basic Authentication ist, dass der Benutzername und das Passwort im Klartext übertragen werden. Dagegen setzt die Digest Access Authentication bei der Übertragung der Zugangsdaten eine Hashfunktion ein, sodass das Passwort nicht rekonstruiert werden kann. Leider sind beide Methoden nicht gegen Man-in-the-middle-Angriff geschützt, was in beiden Fällen zusätzliche Sicherheitsmaßnahmen erfordert.[162]

Die komplette Verschlüsselung der Kommunikation zwischen Client und Server wird dabei als eine ausreichende Maßnahme angesehen. Bereits vor der Übermittlung des Passwortes wird die HTTPS-Verbindung mittels SSL/TLS verschlüsselt, so dass auch das einfache Basic-Verfahren ausreichend sicher ist. Weiterhin kann die Kommunikation gegen den Man-in-the-middle-Angriff durch Einsatz eines Public-Key-Zertifikates gesichert werden, sodass letztendlich eine SSH-equivalente Sicherheitsstufe erreicht werden kann.[163]

5.7 Skalierung

Sollte die Performance des Web Services bei großer Last trotz aller Optimierungen nicht ausreichen, so soll es möglich sein die einzelnen Dienste auf mehrere Server zu verteilen und dadurch die Belastbarkeit zu erhöhen. Im Idealfall sollen folgende ClustWare-Module horizontal skalierbar sein:

- Rails-Webserver
- MySQL-Datenbank
- Memcached-Server
- Appliance-Daemon

Das Framework Rails stellt selbst kein Hindernis bei der Skalierung dar, somit können mehrere Webserver die gleiche Rails-Anwendung ausführen und durch einen Load Balancer koordiniert werden. Die Rails-Anwendung sollte dabei parallelisierbar sein, was

[161] Vgl. [13] S. 433–443.
[162] Vgl. [33] Kapitel 2 Basic Authentication Scheme, Kapitel 3.1.1 Purpose, Kapitel 4.8 Man in the Middle.
[163] Vgl. [34] Kapitel 3.1. Server Identity.

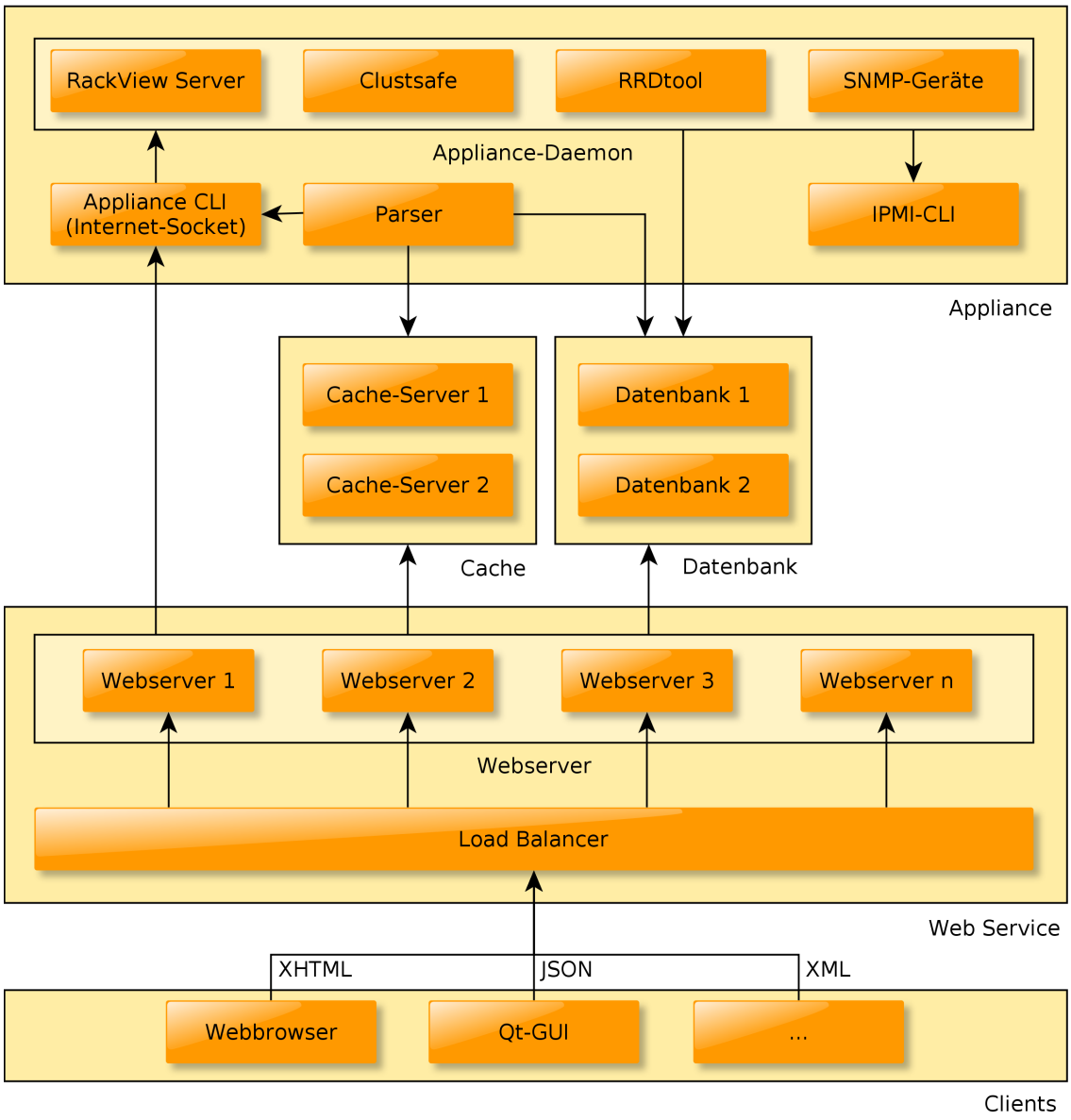

Abbildung 5.7: Dritter Architekturentwurf.

zum Teil schon im Kapitel 5.2 besprochen wurde. Auch die MySQL-Datenbank und der Memcached-Server sind in eine skalierbare Architektur relativ leicht integrierbar, da sie für diesen Zweck entsprechende Mittel bereitstellen.[164]

Es ist schwierig die richtigen Aussagen über die Skalierbarkeit des Appliance-Daemons zu treffen, da das System bei der Entwicklung nicht primär für die Skalierbarkeit ausgelegt war. Dennoch besitzt der Appliance-Daemon eine beachtliche Kapazität, sodass ein Cluster mit schätzungsweise bis zu 10000 Knoten verwaltet werden kann. Wird das Appliance CLI neben dem Unix-Socket auch den Internet-Socket für die Kommunikation zur Verfügung stellen, so wird der Skalierbarkeit nichts mehr im Wege stehen, siehe Abbildung 5.7.

[164] Vgl. [9] Kapitel 2. Concepts of Server Load Balancing, [50] und [16].

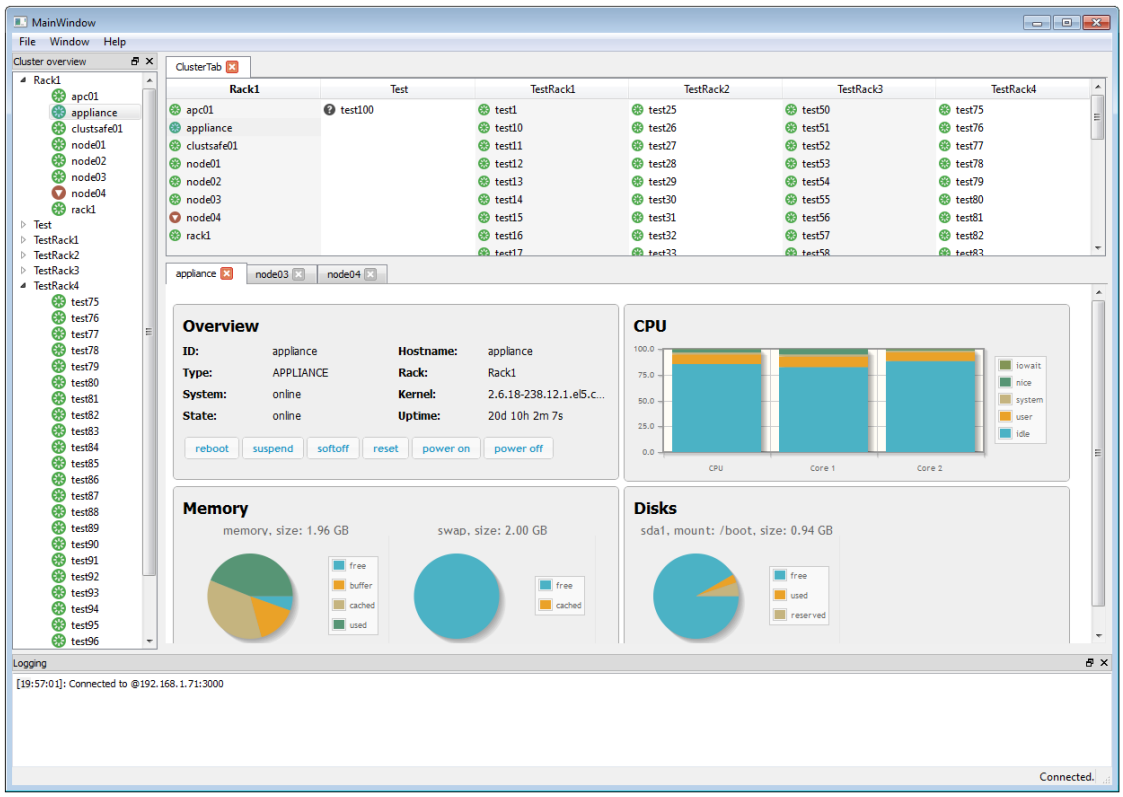

Abbildung 5.8: Qt-Benutzeroberfläche.

5.8 Clients

Die Qt-Benutzeroberfläche und beliebige Webbrowser werden zurzeit als einzige Clients für den entworfenen Web Service angesehen. Da das Web Service die XHTML-Repräsentationen anbietet, ist es naheliegend sie in einem Webbrowser zu betrachten und zu bedienen. Diese Möglichkeit bestand jedoch auch vor der Entwicklung dieses Web Services.

Ein besonderes Interesse stellt die Anbindung an die Qt-Benutzeroberfläche. Die Benutzeroberfläche wurde in dem vorangegangenen Praktikum entwickelt und war für die Kommunikation mit dem Qt-Backend vorgesehen. Durch die Anpassung der Kommunikationsschicht der Benutzeroberfläche lassen sich somit nicht nur die Performance und die Stabilität des Web Services überprüfen, sondern auch die Komplexität und Attraktivität der Entwicklung mit der früheren Lösung vergleichen.

Die Qt-Benutzeroberfläche ist nun als eine hybride Webanwendung zu betrachten. Die Qt-Views werden zu einem Teil direkt vom Qt-Framework generiert (Cluster-Rack-View und Claster-Tree-View) und zum anderen Teil werden die Views durch AJAX aktualisiert und mithilfe des jQuery Frameworks dargestellt (Device-View). Dadurch wird eine maximale Performance erreicht, wobei die Layouts in HTML, JavaScript und CSS definiert werden können, siehe Abbildung 5.8.

5.9 Benchmarks

Zuletzt soll ein API-Benchmark erstellt und ausgewertet werden. Dabei soll der Einsatz des APIs auf unterschiedlich großen Cluster-Systemen analysiert werden, indem die Client-Anfragen und die Cluster-Größe auf einem Test-Cluster simuliert werden.

Seitens der Megware Computer GmbH wurde ein Test-Cluster für die Benchmark-Tests zur Verfügung gestellt. Der Cluster besteht aus 4 Rechenknoten, 2 PDUs und einem Appliance-Knoten. Durch eine Appliance-interne Simulation wurde der Knotenanzahl auf knapp 100 erhöht. Aus diesem Grund ist von einem Test-Cluster mit insgesamt 100 Knoten auszugehen.

Bei der Benchmark sollten die Cluster mit jeweils 1000, 2000, 3000, 4000, 5000, 6000 und 7000 Knoten simuliert werden. Bei der Simulation soll außerdem der Client-Faktor beachtet werden, sodass das Web Service die Anfragen von 5 Clients gleichzeitig beantwortet muss. Dadurch sollte die Umgebung eines Produktiv-Clusters realitätsnah nachgebildet werden.

Für die Erstellung der Benchmark wird das Apache Benchmark Tool eingesetzt. Das Apache Benchmark Tool führt die Benchmark-Tests basierend auf dem HTTP-Protokoll aus, wobei eine definierte Anzahl der Anfragen an eine Webanwendung geschickt wird. Währenddessen wird die Bearbeitungszeit berechnet und analysiert.

Zur Bestimmung der Antwortzeit des Web Services muss eine der Cluster-Größe entsprechende Anzahl der Client-Anfragen generiert werden. Die Anzahl der Client-Anfragen wird wie folgt zu berechnen:

$$Anfragenanzahl_{\text{Simulation}} = \frac{Knotenanzahl_{\text{Simulation-Cluster}}}{Knotenanzahl_{\text{Test-Cluster}}} * Clientanzahl$$

$$Anfragenanzahl_{1000} = \frac{1000}{100} * 5$$
$$= 50$$

Die Menge der übertragenen Daten, die Bearbeitungsdauer auf dem API-Server und die Gesamtantwortzeit auf dem Test-Cluster entsprechen somit den entsprechenden Messdaten auf dem simulierten Cluster, soweit die Repräsentationen mit Paginierung (max. 100 Knoten pro Anfrage) übertragen werden. Soweit alle Test-Parameter bekannt sind, wird der Benchmark-Test durch das Apache Benchmark Tool ausgeführt:

```
# Startet einen Benchmark-Test fuer einen simulierten Cluster
# mit 1000 Knoten mit 5 Clients.
ab -n 50 -c 5 http://10.1.0.200:4000/racks.json
```

Zeit \ Knotenanzahl	1000	2000	3000	4000	5000	6000	7000
90 PSA	0,010	0,010	0,011	0,022	0,015	0,012	0,011
100 PSA	0,017	0,052	0,052	0,067	0,065	0,063	0,054
Gesamt	0,104	0,249	0,362	0,716	0,710	0,775	0,936

PSA – Prozentsatz der Anfragen, die innerhalb einer bestimmten Zeit einzeln bearbeitet wurden.

Tabelle 5.3: Die Web Service Benchmark bei 5 Clients.

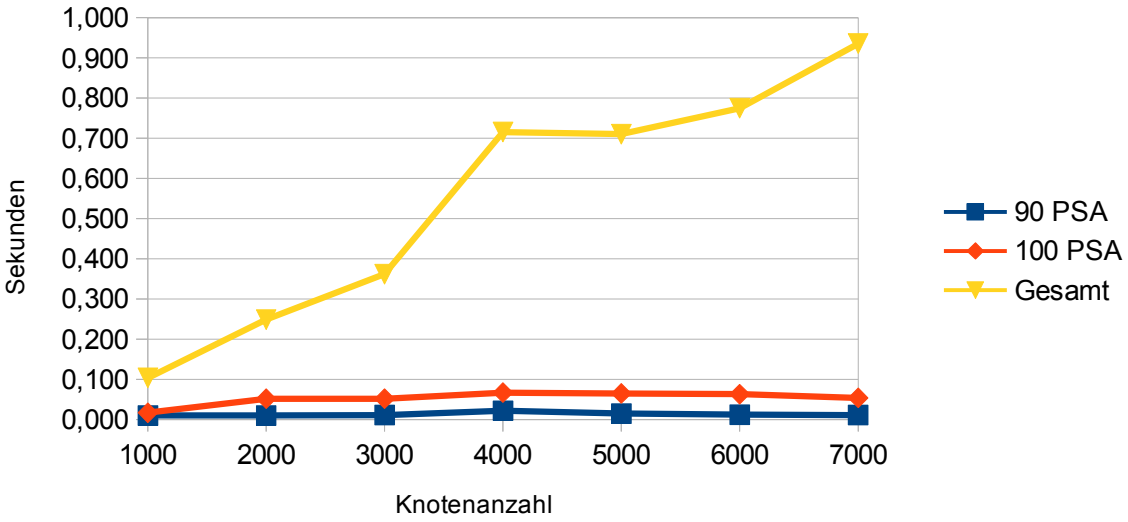

Abbildung 5.9: Die Web Service Benchmark bei 5 Clients.

Nach der Ausführung der Benchmarks werden die Ergebnisse ausgewertet, siehe Tabelle 5.3 und Abbildung 5.9. PSA gibt den Prozentsatz der Anfragen, die innerhalb einer bestimmten Zeit einzeln bearbeitet wurden, an. So wird z. B. die Einzelbearbeitungszeit von 0,010 Sekunden bei 90% aller Anfragen bei einem Cluster mit 1000 Knoten nicht überschritten. Somit ist es erkennbar, dass die Einzelbearbeitungszeit aller Anfragen relativ konstant bleibt. Weiterhin ist die Einzelbearbeitungszeit der Anfragen von der Cluster-Größe unabhängig und übersteigt 0,067 Sekunden nicht.

Die Gesamtbearbeitungszeit aller Anfragen ist hingegen von der Cluster-Größe direkt linear abhängig. Trotz dieser Abhängigkeit kann man von einer hohen Performance ausgehen, da beim Cluster mit 7000 Knoten alle 5 Client-Anfragen innerhalb einer Sekunde beantwortet werden können.

6 Fazit und Ausblick

Das Ziel dieser Arbeit war es, ein RESTful Web Service API zu erarbeiten. Dabei sollte die neue Architektur nicht nur eine hohe Performance, sondern auch eine bessere Wartbarkeit und Plattformunabhängigkeit im Vegleich zu Qt-basierten Lösung aufweisen. Weiterhin sollte der Entwicklungsprozess der auf dem Web Service aufbauenden Endanwendungen effizienter gestaltet werden.

Als Einführung in das Thema wurden die Prinzipien des REST-Architekturstils, der Aufbau und die Funktionsweise eines Clusters sowie der modulare Aufbau der Megware Cluster-Managementsoftware näher betrachtet. Weiterhin wurde das Ruby on Rails Framework als Grundlage für die Implementierung untersucht.

Der REST-Architekturstil wurde als Fundament zur Realisierung der neuen Architektur ausgewählt. Aufgrund der Vermutung, dass diese Architektur kaum an die Leistungsfähigkeit einer Qt-basierten Lösung heranreichen kann, wurde beim Entwurf besonders großer Wert auf die Performance und Skalierbarkeit des APIs gelegt.

Darauf aufbauend wurde das Web Service API ansatzweise implementiert, wobei die einzelnen Programmschichten wie Parser, Modell, Controller und View durch alle Ebenen hindurch umgesetzt wurden. Weiterhin wurde die Qt-Benutzeroberfläche an die neue Architektur angepasst, was die Analyse der Performance und die Bewertung der Praxistauglichkeit ermöglichte. Die positive Resonanz und die kurze Entwicklungszeit zeigten dabei, dass bei der Qt-Benutzeroberfläche der richtige Ansatz gewählt wurde und die Entwicklung durch die Umstellung auf das Web Service API merkbar vereinfacht wurde.

Zuletzt wurden die Benchmarks ausgewertet. Diese haben gezeigt, dass das Web Service API eine hohe Performance aufweist, was durch die Skalierung und Optimierung der bestehenden Appliance-Architektur erreicht wurde. Damit wurde bestätigt, dass das RESTful Web Service API alle am Anfang formulierten Anforderungen erfüllt.

Gleichzeitig haben die Benchmark-Tests und die Anbindung der Qt-Benutzeroberfläche auch die momentanen Grenzen des API gezeigt. So muss z. B. die Parser-Stabilität verbessert und das API durch weitere Funktionalität wie Benutzerverwaltung erweitert werden.

Im Hinblick auf die Zukunft bietet sich an, dieses API bei der Entwicklung von neuen Benutzeroberflächen für Smartphones und Tablets einzusetzen. Zurzeit wird auch an einer neuen RackView-Benutzeroberfläche gearbeitet, die auf das entwickelte API mit hoher Wahrscheinlichkeit zugreifen wird. Somit kann man von einer Weiterentwicklung des RESTful Web Service API ausgehen.

Anhang A: Qualitätsmerkmale nach ISO 9126

Qualitätsmerkmale	wichtig	normal	unwichtig	Notizen
Funktionalität		X		
• Richtigkeit		X		
• Interoperabilität		X		
• Sicherheit		X		
• Ordnungsmäßigkeit			X	
Zuverlässigkeit		X		
• Reife			X	
• Fehlertoleranz	X			Absturzsicherheit
• Konformität			X	
Benutzbarkeit		X		
• Verständlichkeit	X			
• Erlernbarkeit		X		
• Bedienbarkeit	X			
• Attraktivität		X		
Effizienz	X			
• Zeitverhalten	X			Antwortzeit 5-10 Sek.
• Verbrauchsverhalten	X			CPU-Entlastung
Änderbarkeit		X		
• Analysierbarkeit	X			
• Modifizierbarkeit	X			
• Stabilität		X		
• Testbarkeit		X		
Übertragbarkeit		X		
• Anpassbarkeit		X		
• Installierbarkeit		X		
• Koexistenz	X			
• Austauschbarkeit		X		

Anhang B: Verwendete Werkzeuge

Entwicklungsumgebung:

- Ubuntu 11.10
- Microsoft Windows 7
- Ruby 1.9.3
- Ruby on Rails 3.1.0
- Nokia Qt SDK 4.7.1
- Nokia Qt Creator 2.3.1
- Oracle MySQL Workbench 5.2.36
- JetBrains RubyMine 3.2.4

Dokumentation:

- LaTeX und LaTeX-Vorlagen von Prof. Klaus Dohmen
- Texmaker 3.1
- Doxygen 1.7.4
- RDoc 3.11
- yWorks yEd 3.8

Test-Umgebung:

- Megware Test-Cluster (4 Rechenknoten, 2 PDUs, 1 Appliance-Knoten, 1 Rack-View, 1 Rack-Management-Modul)
- Thin Ruby Webserver 1.3.0
- Apache Benchmark Tool 2.0.40

Literatur- und Quellenverzeichnis

[1] ALLAMARAJU, S.: *RESTful Web Services Cookbook*. O'Reilly Media, Sebastopol, 2010.

[2] AMUNDSEN, M.: *Supporting PUT and DELETE with HTML FORMS*. http://amundsen.com/examples/put-delete-forms/, Stand: 04.10.2011.

[3] ANDROID-LIGHTHOUSE COMMUNITY: *Qt for Android*. http://code.google.com/p/android-lighthouse/, Stand: 19.08.2011.

[4] APIDOCK, NODETA: *Authenticate or request with http basic*. http://apidock.com/rails/ActionController/HttpAuthentication/Basic/ControllerMethods/authenticate_or_request_with_http_basic, Stand: 3.12.2011.

[5] BAUKE, H.; MERTENS, S.: *Cluster Computing*. Springer-Verlag, Berlin Heidelberg, 1. Auflage, 2006.

[6] BAYER, T.: *REST Web Services*. Webseite von Orientation in Objects GmbH. http://www.oio.de/public/xml/rest-webservices.htm, Stand: 19.08.2011.

[7] BELLER, S.: *Praktikumsbericht: Entwicklung einer Qt-basierten grafischen Benutzeroberfläche für die Megware Cluster-Managementsoftware*. Technischer Bericht, Hochschule Mittweida, 2011.

[8] BONGERS, F.; VOLLENDORF, M.: *jQuery. Das Praxisbuch*. Galileo Press, Bonn, 2010.

[9] BOURKE, T.: *Server Load Balancing*. O'Reilly Media, Sebastopol, 2001.

[10] CARL, D.: *Praxiswissen Ruby on Rails*. O'Reilly Verlag, 2007.

[11] CARNEIR, C.; AL BARAZI R.: *Beginning Rails 3*. Apress, 2010.

[12] FAUSER, C.; MACAULAY, J.; OCAMPO-GOODING E.; GUENIN J.: *Rails 3 in a Nutshell*. O'Reilly Verlag, 2010. http://ofps.oreilly.com/titles/9780596521424/, Stand: 29.09.2011.

[13] FERNANDEZ, O.: *THE RAILS 3 WAY*. Addison-Wesley, 2011.

[14] FIELDING, R. T.: *Architectural Styles and the Design of Network-based Software Architectures*. Doktorarbeit, UNIVERSITY OF CALIFORNIA, IRVINE. http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm, Stand: 19.08.2011.

[15] FISCHER, L.: *A Beginner's Introduction to HTTP and REST*. Artikel bei Nettuts+. http://net.tutsplus.com/tutorials/other/a-beginners-introduction-to-http-and-rest/, Stand: 19.08.2011.

[16] FITZPATRICK, B.: *Distributed Caching with Memcached*. http://www.linuxjournal.com/article/7451, Stand: 3.12.2011.

[17] FLANAGAN, D.; MATSUMOTO Y.: *The Ruby Programming Language*. O'Reilly, Sebastopol, 2008.

[18] HARTL, M.: *Ruby on Rails 3 Tutorial*. Addison-Wesley Longman, Amsterdam, 2010.

[19] JOHNSON, R.; GAMMA, E.; HELM-R.; VLISSIDES J.: *Entwurfsmuster: Elemente wiederverwendbarer objektorientierter Software*. Addison-Wesley, München, 2004.

[20] JSON-DEVELOPERS: *Einführung in JSON*. http://www.json.org/json-de.html, Stand: 3.12.2011.

[21] KATZ, Y.; BIGG R.: *Rails 3 in Action*. Manning Pubn, Shelter Island, 2011.

[22] KERSKEN, S.: *IT-Handbuch für Fachinformatiker*, Kapitel 13. Galileo Computing, 4. Auflage, 2009. http://openbook.galileocomputing.de/it_handbuch/fachinformatiker_13_lamp_001.htm, Stand: 19.08.2011.

[23] LIEBEL, O.: *Linux Hochverfügbarkeit*. Galileo Press, Bonn, 2011.

[24] LOPEZ, B.: *YAJL C Bindings for Ruby*. https://github.com/brianmario/yajl-ruby, Stand: 2.12.2011.

[25] MANHART, DR. K.: *TecChannel: Networked Computing – Grundlagen und Anwendungen*. http://www.tecchannel.de/netzwerk/wan/439222/networked_computing_grundlagen_und_anwendungen/, Stand: 19.10.2011.

[26] MEGWARE COMPUTER GMBH: *ClustWare-Appliance Dokumentation*. unveröffentlichte Entwicklerdokumentation, Stand: 25.10.2011.

[27] MEGWARE COMPUTER GMBH: *ClustWare Benutzerhandbuch*. unveröffentlichte Dokumentation, August 2011.

[28] MEMCACHED-DEVELOPERS: *Memcached – a distributed memory object caching system*. http://memcached.org/, Stand: 2.12.2011.

[29] MEMCACHED-DEVELOPERS: *Memcached Wiki*. http://code.google.com/p/memcached/wiki/NewStart?tm=6, Stand: 2.12.2011.

[30] MERCIER, C.: *YAML, ZAML, Marshal & JSON benchmark in Ruby*. http://blog.carlmercier.com/2010/11/19/yaml-zaml-marshal-json-benchmark-in-ruby/, Stand: 2.12.2011.

[31] MORSY, H.; OTTO, T.: *Ruby on Rails 2*. Galileo Computing, 2008. http://openbook.galileocomputing.de/ruby_on_rails/, Stand: 19.09.2011.

[32] NETWORK WORKING GROUP: *RFC 2616 – Hypertext Transfer Protocol – HTTP/1.1*. http://tools.ietf.org/html/rfc2616, Stand: 19.08.2011.

[33] NETWORK WORKING GROUP: *RFC 2617 – HTTP Authentication: Basic and Digest Access Authentication*. http://tools.ietf.org/html/rfc2617, Stand: 05.12.2011.

[34] NETWORK WORKING GROUP: *RFC 2818 – HTTP Over TLS*. http://tools.ietf.org/html/rfc2818, Stand: 05.12.2011.

[35] NETWORK WORKING GROUP: *RFC 3986 – Uniform Resource Identifier (URI): Generic Syntax*. http://tools.ietf.org/html/rfc3986, Stand: 19.08.2011.

[36] NOKIA CORPORATION: *Supported Platforms*. http://developer.qt.nokia.com/wiki/Platform_Support, Stand: 19.08.2011.

[37] ORSINI, R.: *Rails Kochbuch*. O'Reilly Verlag, 2007.

[38] QT-IPHONE COMMUNITY: *Qt-iPhone*. http://www.qt-iphone.com/, Stand: 19.08.2011.

[39] RAILSCASTS: *Authentication in Rails 3.1*. http://railscasts.com/episodes/270-authentication-in-rails-3-1, Stand: 3.12.2011.

[40] RAILSCASTS: *HTTP Basic Authentication*. http://railscasts.com/episodes/82-http-basic-authentication, Stand: 3.12.2011.

[41] RAILSGUIDES: *Action Pack — On rails from request to response*. http://ap.rubyonrails.org/ Stand: 19.08.2011.

[42] RAILSGUIDES: *Getting Started with Rails*. http://guides.rubyonrails.org/, Stand: 19.08.2011.

[43] RAILSGUIDES: *Ruby on Rails v3.1.1 API Doc*. http://api.rubyonrails.org/ Stand: 19.08.2011.

[44] RAUBER, T.; RÜNGER, G.: *Parallel Programming*. Springer-Verlag, Berlin Heidelberg, 1. Auflage, 2010.

[45] RICHARDSON, L.; RUBY, S.: *RESTful Web Services*. O'Reilly Media, Sebastopol, 2007.

[46] RODRIGUEZ, A.: *RESTful Web services: The basics*. Webseite von IBM developerWorks. http://www.ibm.com/developerworks/webservices/library/ws-restful/, Stand: 19.08.2011.

[47] RUBY, S.; THOMAS, D.; HEINEMEIER HANSSON D.: *Agile Web Development with Rails*. The Pragmatic Bookshelf, 2011.

[48] SKORKIN, A.: *Serializing (And Deserializing) Objects With Ruby*. http://www.skorks.com/2010/04/serializing-and-deserializing-objects-with-ruby/, Stand: 2.12.2011.

[49] SLAGELL, M.: *Ruby User's Guide*. http://www.rubyist.net/~slagell/ruby/, Stand: 19.08.2011.

[50] SUN MICROSYSTEMS, MYSQL: *Leistungsoptimierung für das Datenbanksystem MySQL Cluster*. http://krsteski.de/wp-content/uploads/2010/09/MySQL_Cluster_Perfomance_WP_deutsch.pdf, Stand: 3.12.2011.

[51] THOMAS, D.; FOWLER, C.; HUNT A.: *Programming Ruby 1.9*. Pragmatic Programmers, 2009.

[52] TILKOV, S.: *REST - Der bessere Web Service?* Artikel bei JAXenter. http://it-republik.de/jaxenter/artikel/REST---Der-bessere-Web-Service-2158.html, Stand: 19.08.2011.

[53] TILKOV, S.: *REST und HTTP*. dpunkt.verlag GmbH, Heidelberg, 2009.

[54] TILKOV, S.: *RESTful Web Services mit Rails*. RailsWay, 2009. http://www.innoq.com/files/RM_1.09_tilkov_REST.pdf, Stand: 19.09.2011.

[55] W3C COMMUNICATIONS TEAM: *Web Services Architecture*. http://www.w3.org/TR/ws-arch/, Stand: 3.10.2011.

[56] W3C COMMUNICATIONS TEAM: *XML in 10 points*. http://www.w3.org/XML/1999/XML-in-10-points.html.en, Stand: 3.12.2011.

[57] WEBBER, J.; PARASTATIDIS, S.; ROBINSON I.: *REST in Practice*. O'Reilly Media, Sebastopol, 2010.

[58] WINTERMEYER, S.: *Ruby on Rails 3.0 und 3.1*. http://www.ruby-auf-schienen.de/, Stand: 19.08.2011.

Erklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und nur unter Verwendung der angegebenen Literatur und Hilfsmittel angefertigt habe.

Stellen, die wörtlich oder sinngemäß aus Quellen entnommen wurden, sind als solche kenntlich gemacht.

Diese Arbeit wurde in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegt.

Mittweida, 13. Dezember 2011